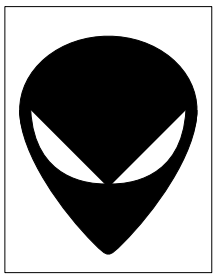
THE UNIVERSITY OF MANCHESTER
DEPARTMENT OF COMPUTER SCIENCE

THIRD-YEAR PROJECT REPORT (1997)

UFOLEX AND UFOYACC

MARK LOMAS

Supervisor: Dr. Chris Kirkham



Dedicated to Marian Lomas

Abstract

UFOLex and UFOYacc

Author: Mark Lomas

UFO [1] is a new parallel programming language, combining the best of modern ideas about the functional and objected-oriented paradigms. Lex and Yacc [5] are standard Unix language processing tools which generate code suitable for use in a C compiler. The aim of this project is to produce versions of Lex and Yacc written in UFO, capable of generating code suitable for use in a UFO compiler.

Important aspects of the design of UFOLex and UFOYacc will be discussed. Problematic implementation issues will be presented including an assessment of the suitability of UFO as an implementing language. In the conclusion, the practicality of writing a UFO compiler using UFOLex and UFOYacc will be considered.

Supervisor: Dr. C Kirkham

Contents

1. Introduction
- 1.1 Aim of project 1
- 1.2 'United Functions and Objects' background 1
- 1.3 Lex and Yacc background 1
- 1.4 Motivation 2
- 1.5 Outline of Report 3
- 1.6 History 3

2. UFOLex and UFOYacc Overview
- 2.1 Organisation 4

3. UFOLex
- 3.1 Role of UFOLex 6
- 3.2 Operation of UFOLex 6
- 3.3 UFOLex features 6
- 3.4 UFOLex programs 7
- 3.5 Regular expressions 8
- 3.6 Context Sensitivity 8
- 3.7 UFOLex definitions 8
- 3.8 Start State definitions 9
- 3.9 UFOLex rules 9
- 3.10 Action code 10
- 3.11 Start state lists 10
- 3.12 UFOLex input program summary 11
- 3.13 Parser-Lexer communication 11
- 3.14 Changing start state from within an action 13
- 3.15 Lexer conflicts 13
- 3.16 Lexical Errors 14
- 3.17 Input from strings and buffers 15
- 3.18 Automatic line number counting 15

4. UFOYacc
- 4.1 Role of UFOYacc 16

Contents

4.2 Operation of UFOYacc 16
4.3 UFOYacc features 17
4.4 UFOYacc programs 17
4.5 Grammar rules 17
4.6 UFOYacc definitions section. 18
4.7 Terminals and Nonterminals 19
4.8 Grammar disambiguation in Shift/Reduce parsing 19
4.9 Assigning precedence and associativity. 20
4.10 S-attributed syntax directed translation and user action code 22
4.11 Start symbol 23

5. UFOLex Design
5.1 Design overview 24
5.2 Lexical analyser breakdown 24
5.3 Finite automaton simulator 25
5.4 Choice of finite automata generator algorithms 25
5.5 Regular expressions 26
5.6 Language operations and regular expressions 26
5.7 Conversion of notational shorthands 26
5.8 DFA construction algorithm 27
5.9 Syntax tree constituent components 28
5.10 Decoration of syntax tree with position information 30
5.11 Scanning and parsing a UFOLex program 31
5.12 Constructing the DFA 32
5.13 Subset and transition table construction 32
5.14 Incorporating multiple regular expressions 33
5.15 Simulator enhancements 35
5.16 Incorporating start states 36
5.17 Trailing Context 37

6 UFOYacc design
6.1 The LR parsing algorithm 39
6.2 Parsing of a UFOYacc program 39
6.3 Construction of canonical LR items 40
6.4 Construction of Action and Goto tables 40
6.5 Executing S-attributed action code 41

Contents

7. Conclusions
7.1 Achievements 42
7.2 UFO as an implementation language 42
7.3 UFO language considerations 42
7.4 Unimplemented aspects 42
7.5 The next step - a UFO compiler 43

References 44

Appendix A 45
A.1 Example UFOLex input program 45
A.2 Example lexer generated by UFOLex from A.1 45
A.3 Example UFOYacc input program 50
A.4 Example parser created by UFOYacc from A.3 51

Appendix B 57
B.1 Regular expressions 57
B.2 A collected syntax of the UFOLex input program 59
B.3 Translation of notational shorthands in regular expressions 60
B.4 BNF grammar for UFOLex programs 61
B.5 UFOLex's BNF grammar for recursive decent parsing 64

Appendix C 68
C.1 A collected syntax of the UFOYacc input program 68
C.2 UFOYacc's BNF grammar for recursive decent parsing 70

Appendix D 72
D.1 Lex and Yacc - further motivations 72

Appendix E 73
E.1 Regular grammar property 73
E.2 Left context 73
E.3 Right context 73

Appendix F 75
F.1 UFO1.0/S compiler problems 75

1. Introduction

1.1 Aim of project

Lex and Yacc are language processing tools developed primarily to assist with the construction of compilers. They generate C code of a suitable standard for inclusion into the front end of a compiler. The aim of this project was to develop versions of Lex and Yacc which would be suitable for use with a compiler for a new programming language developed in the Department of Computer Science at the University of Manchester. They would generate UFO code of a suitable standard for inclusion into a UFO compiler.

1.2 'United Functions and Objects' background

'United Functions and Objects' (UFO) [1] is a parallel programming language. It's aim is to make the task of programming parallel computers much easier, whilst fully exploiting the power that they provide. Unlike languages such as FORTRAN, all parallelism occurs without the programmer having to expend any effort.

UFO incorporates the best of modern ideas about parallelism and programming paradigms. It combines the properties of modern functional and object-oriented languages. It incorporates state through the notion of stateful objects and hence is not a pure functional language. This allows a full range of key parallel features to be supported.

Despite this, the first stable implementation, UFO1.0/S, is sequential and currently lacks implementation of certain language features.

1.3 Lex and Yacc background

Lex and Yacc are standard Unix compiler construction tools which successfully solve the general problem of lexical and syntactic analysis. However their use isn't limited to just compilers. The construction process for pretty printers, word processors and interpreters could all benefit from the use of Lex and Yacc.

Lex and Yacc are concerned with the recognition of languages. They help the compiler writer, to write programs that transform structured input and perform actions. They allow the programmer to make a language description explicit and to concentrate on WHAT to do with

a recognised language, not HOW to recognise it.

Computer applications frequently need to logically divide their input into meaningful units, known as '*lexemes*'. For example, a compiler needs to divide a user program into identifiers, numbers and reserved words. Lex helps to automate this task, which is commonly referred to as lexical analysis or '*lexing*' for short.

It is commonly the case that applications only use a lexical analyser[1] to assist in achieving the more difficult task of syntactic analysis. Syntactic analysis is the process of grouping lexemes and rejecting illegal combinations of them. Yacc automatically generates syntactic analysers[2] to help with this task, which is commonly referred to as '*parsing*'. A syntax analyser is made much simpler if a lexer has first divided the user program into meaningful lexemes and separated any important information from irrelevant information. As an example, consider the task of checking for valid assignment statements in a C program. The compiler must only recognise an assignment statement if the identifiers and numbers in the assignment are given conforming to a predefined grammar or syntax.

1.4 Motivation

UFO is very new. There are very few full scale applications implemented in UFO. The key motivation behind this project is to experiment with the UFO language through the development of UFOLex and UFOYacc, in an attempt to learn more about it. It is important to point out that these tools will not just be re-implementations of existing versions of Lex and Yacc. The emphasis will lie with finding the advantages and disadvantages of using UFO as the implementation language, or, for example, with deciding where to use the notion of state and why.

Further motivation arises from the fact that although the UFO1.0/S compiler is written mostly in UFO, its front end is written in Haskell, an aspect which one would wish to avoid in future releases of the compiler. The hope is that, UFOLex and UFOYacc could be of considerable use in this area. In addition, we need UFOLex and UFOYacc because currently there are no lexical or syntactic analyser generators for the UFO programming language.

For further motivations see Appendix D.1 which is concerned with the motivations behind using automatic parser and lexer generators.

1. Lexical analysers are often referred to as 'lexers'.
2. Syntactic analysers are often referred to as 'parsers'.

1.5 Outline of Report

This report will concentrate on each utility in turn. A general overview of how the final tools should be used, will be followed by descriptions of the design of each tool. This stage will highlight key decisions and major design issues, leaving the next section to describe the interesting aspects of the implementation. In the conclusion, the achievements of the project will be discussed in addition to focusing on the ease of implementation and those advantages and disadvantages of using UFO1.0 as the implementation language. Finally, areas where UFOLex and UFOYacc would need to be built upon in order to address the practicalities of constructing a UFO compiler will be mentioned.

1.6 History

The very first versions of Lex and Yacc were developed in the 1970's by AT&T. Oddly enough, yacc was developed first and Lex was developed soon after by an undergraduate summer intern. They were UNIX tools specific to the 'C' programming language, although Lex could also generate scanners for the RATFOR programming language.

More recent implementations of these utilities include Berkely Yacc and GNU Bison. Berkely Yacc is almost an exact re-implementation of AT&T yacc with a few extra features. GNU Bison is compatible with AT&T yacc, although there are occasional yacc grammars that don't work properly. Bison is derived from an early version of Berkeley yacc but has now been developed independently for several years. Flex is based on AT&T lex, but is much more reliable, generates faster scanners and is free from some restrictions. It is highly compatible with AT&T lex.

Even more recently, versions of Lex and Yacc have been developed for different operating systems, for example MKS Lex/Yacc and Abraxas Lex/Yacc. The main differences between these versions and AT&T's versions are to do with running under different operating systems such as MS-DOS and OS/2.

Some versions of Lex and Yacc have been developed for use with programming languages other than 'C' . ML-Lex developed at Princeton University and ML-Yacc developed at Princeton University and Carnegie Mellon University, give the ML programmer access to these powerful utilities.

2. UFOLex and UFOYacc Overview

2.1 Organisation

The UFOLex compiler takes as input, a language for specifying a lexical analyser. This consists of a series of regular expressions and actions. This is known as a '*Lex program*'. UFOLex then outputs a tabular representation of a transition diagram constructed from the regular expressions, together with a standard algorithm that uses the table to recognise lexemes and perform actions. This output takes the form of a file containing program fragments suitable for inclusion into an application.

The UFOYacc compiler takes as input, a language for specifying a syntactic analyser. This consists of a series of BNF like recursive grammar specifications and actions. This is known as a '*Yacc program*'. In a similar way to UFOLex, UFOYacc outputs a file containing program fragments suitable for inclusion into an application.

The resulting output from Lex and Yacc can be used in combination to solve a language recognition problem. Both Lex and Yacc thus provide program fragments that are easily combined. fig. 2.1.1 summarises this operation.

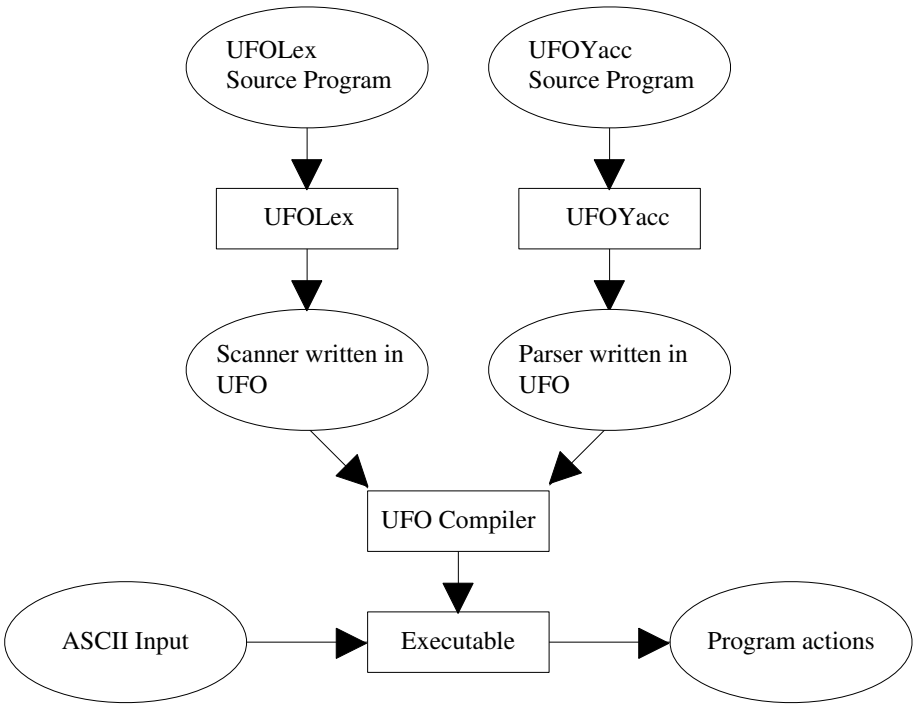


Figure 2.1.1 Typical usage of UFOLex and UFOYacc

Fig. 2.1.1 also shows a resulting parser and a lexer being compiled by the UFO compiler to produce and executable. It also shows the result of running the executable given a stream of ASCII input characters. As an example, a typical compiler written using UFOLex and UFOYacc would use the 'program actions' to build parse trees, perform typechecking, carry out optimisations and generate machine code,

Within the applications executable, we can view the lexer and the parser as two cooperating entities which must communicate with each other in an attempt to allow actions to be performed. Fig. 2.1.2 illustrates this arrangement. This is the organisation of the front end of a typical compiler. Messages passing between the lexer and the parser are known as '*Lexical Units*'. A '*Lexical Unit*' consists of a token uniquely identifying the type of lexeme found and an associated attribute value. This attribute value is used to communicate other important information about the lexeme to the parser.

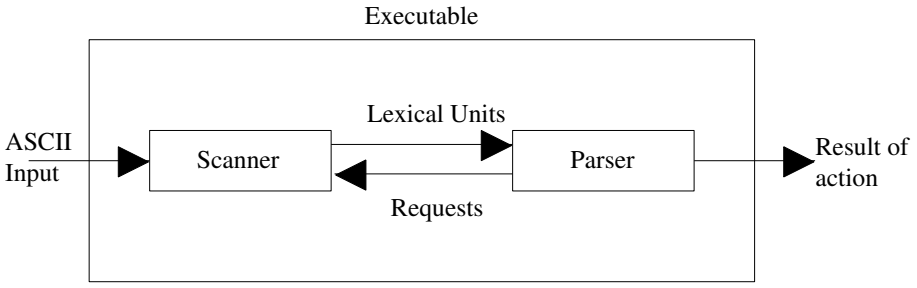


Figure 2.1.2 Cooperation of UFOLex and UFOYacc program fragments

UFOLex and UFOYacc are not confined in such a way that they have to be used together. Infact each utility can be used independently. This is often appropriate for tasks which aren't complex enough to warrant the use of both utilities. For example, why use a syntax analyser to implement a pretty printer, when just a lexical analyser will suffice?

UFOLex and UFOYacc can be viewed as compilers themselves. Original versions of Lex and Yacc were implemented in languages such as C producing program fragments in C. UFOLex and UFOYacc are versions written in UFO, which produce UFO program fragments.

3. UFOLex

3.1 Role of UFOLex

UFOLex automates the construction of lexical analysers. It can utilise the most efficient pattern matching algorithms and thus create efficient lexical analysers. We use grammars to represent valid sentences in a language and we use 'automata' to recognise whether a given sentence is in a language. UFOLex generates an efficient finite automaton (FA) recogniser[1] for a given set of regular expressions. It's underlying function is to execute actions triggered by the matching of patterns in input strings.
Common tasks for the finite automaton include:

1. Stripping unnecessary white space e.g. newlines, tabs and spaces
2. Helping in error message generation by providing line number information
3. To perform pre-processor activities such as including input strings from other input sources.

3.2 Operation of UFOLex

UFOLex operates in a similar fashion to Unix Lex/Flex utilities, providing a mechanism for solving lexical analysis problems. It takes a UFOLex source program via the command line ('.lex') and generates a '.ufo' module containing the resulting scanner. The scanner and the application are easily combined by using the UFO compiler to link them together. See Appendices A.1 and A.2 for an example of a UFOLex input program and it's resulting lexer generated by UFOLex.

3.3 UFOLex features

Below is a list of issues which UFOLex must deal with in a suitable way, in order to make it as useful as possible to the compiler writer. Each of these features relates to some aspect of a UFOLex input program.

1. Regular definitions (substitutions)
2. Start States & changing start states in action code
3. Rules & return values from associated action code
4. Regular expressions

1. See reference [3], sec. 3.6, p.113-p.117.

5. Context Sensitivity (left & right)
6. Unmatched Input
7. File nesting / chaining
8. Input from strings and buffers
9. Automatic line number counting
10. Lexer conflicts
11. Parser-scanner communication
12. Lexical errors

Using the UFOLex input program we can specify what we want UFOLex to do and how we want it to behave.

3.4 UFOLex programs

A UFOLex program is based largely on the syntax used to describe the ML-Lex and Flex input programs[1]. It uses regular expressions, which are a notation used to specify patterns. All tokens can be found by matching their patterns with the input string.

The UFOLex source program contain a series of rules which consist of regular expressions and associated actions, written as UFO program fragments. Other UFO code required by the scope of code used in the actions can be placed in a special user code section at the beginning of the UFOLex source program. This code, along with the action code is copied verbatim to the generated scanner[2]. Finally there exists a section in the UFOLex program, where the user can tell UFOLex how to behave. Vital information required by UFOLex for interpreting the rules section of the input program should be given here.

```
USER DECLARATIONS
%%
UFOLEX DEFINITIONS
%%
RULES
```

Figure 3.4.1 UFOLex input program breakdown

Fig. 3.4.1 gives a summary of the content of the UFOLex input program whilst outlining the syntax used. The three main sections of the source program are specified by separating them with a '%%' delimiter. As a consequence, '%%' can not appear anywhere in the user

1. Flex uses notational conveniences which make the specification of exclusive start states much easier.
2. UFOLex is not concerned with whether or not it is correct.

declarations section, i.e. within UFO comments or strings. See Appendix A.1, for an example UFOLex program of this structure.

3.5 Regular expressions

Regular expressions are a simple language for denoting classes of strings. A regular expression allows us to define a language precisely in terms of a set of basic operations on a vocabulary. For the purposes of UFOLex, the vocabulary is the ASCII character set. To provide greater usability as a language, versions of Lex support a standard set of notational shorthands. UFOLex adopts this standard too. See Appendix B.1, for a full description of the particular regular expression dialect used by UFOLex to support pattern matching.

3.6 Context Sensitivity

Regular expressions in UFOLex are able to specify patterns that are sensitive to left and right context, i.e. text that precedes or follows the token. This is one of the properties of the two classes of context sensitive grammar, the regular and context-free grammars. There are several techniques at the user's disposal for exploiting left and right context sensitivity in UFOLex. Appendices E.2 and E.3 describe them all.

3.7 UFOLex definitions

The user can use the definitions section of the UFOLex input program to assign names to regular expressions which will be used in the rules section. These will be referred to as '*Regular Definitions*' and provide a notational shorthand which simplifies complex regular expressions. Regular definitions are specified using the same syntax as ML-Lex:

```
identifier = regular expression ;
```

Example:

```
digit = [0-9];
```

In a regular expression, patterns may contain references to regular definitions by placing their name in braces[1] e.g. `{digit}`. Regular expressions are further addressed when describing the UFOLex rules section.

1. These are commonly referred to as 'substitutions'

To ensure portability, such substitutions are treated as though the pattern corresponding to the regular definitions name is surrounded by parentheses. This is because in a few versions of Lex, this is not the case and it can result in a regular expression with a very different meaning. For example, in fig. 3.7.1, '`{def}+`' could either be interpreted as '`(abc)+`' or '`abc+`' depending on which version of Lex is being used.

```
def = abc;
%%
{def}+
```

Figure 3.7.1 UFOLex program snippet highlighting a possible portability problem

3.8 Start State definitions

The definitions section of the UFOLex input program, is also host to start state definitions. These are used in the rules section to tell the lexical analyser which patterns described by regular expressions should be used to match the input string at a given time. '*Exclusive*' start states provided by UFOLex, differ from those provided in the original AT&T Lex specification. The 'exclusivity' means that a rule defined upon start state X is matched when in state X, but a rule not defined upon X is not matched. With the AT&T Lex specification, rules with the second condition are matched too. '*Exclusive*' start states are much more useful and are used by Flex. The syntax for specifying new start states is as follows:

```
%s identifier_list ;
```

or

```
%S identifier_list ;
```

The '*identifier_list*' is just a sequence of identifiers separated by spaces.

3.9 UFOLex rules

A rule is made up of a regular expression (part of which may have been defined earlier in the UFOLex definitions section), an action, delimited by a '=>' token and preceded by an optional start state list. Fig. 3.9.1 shows an example of a rule which could be used to match an identifier, or variable name in a programming language.

```
[A-Za-z][A-Za-z0-9]*  =>  { return IDENTIFIER(yytext) }
```

Figure 3.9.1 Example UFOLex rule

3.10 Action code

The action code of a rule is a standard UFO code fragment with the only restriction that it is of the form consisting of curly braces and a '`return`' keyword. All action code fragments should return a value of a fixed type. This is because the resulting automaton produced by UFOLex will take the form of a function with a fixed return value type. A change of start state can be made from within an action. For a complete description of the UFO1.0/S language see [2], the UFO web site.

3.11 Start state lists

A start state list for a rule is optional. If it is omitted, then the rule applies only when the automaton is in the 'INITIAL' start state. The automaton always starts out in this state and can return to it at any time.

A start state list precedes a rule or a list of rules and is delimited by angled brackets. It consists of a list of comma separated start symbols, declared in the UFOLex definitions section. It tells UFOLex when a particular rule is active. If the automaton is in any one of the states given in the start state list, then the rule is active and is considered for matching. Individual rules can be preceded by their own list of start states as shown in fig. 3.11.1.

```
<start state list> regular_expression  =>  {...code...}
<start state list> regular_expression  =>  {...code...}
...
...
etc.
```

Figure 3.11.1 Syntax of rules incorporating start state lists

To help reduce verbose descriptions, UFOLex also adopts a syntax used by 'Flex' to describe a group of rules which have the same start state list. Thus, a UFOLex rules section can take the form given in fig. 3.11.2.

```
<start state list> (
                        regular expression  =>  {...code...}
                        regular expression  =>  {...code...}
                        regular expression  =>  {...code...}
                        ...
                        etc.
)
```

Figure 3.11.2 Grouping of rules under a common start state list.

A set of parentheses is used, to enclose the desired region of rules. Any rules defined in the scope of the parentheses, will only be recognised if the current state of the automaton is a member of the associated start state list.

3.12 UFOLex input program summary

The syntax of a UFOLex input program is best described using a more formal notation. Appendix B.2 gives an EBNF description of the syntax. Only a few important operational characteristics have yet to be described. Having presented the key features of the UFOLex input program, we can move on to look at how common user requirements are catered for by UFOLex.

3.13 Parser-Lexer communication

When a UFOLex lexer and a UFOYacc parser are used together, the parser acts as the higher level routine. It calls the lexer routine whenever it needs a token from the input. The lexer then scans through the input until it finds a match to a pattern. It returns the token, to the parser, which the lexer and the parser have to agree on. We resolve the problem by letting the parser define them.

Not all tokens are of interest to the parser. A common example is that of whitespace. A parser is much simpler if whitespace is ignored. In such a case, the lexer doesn't return so that it can continue onto the next token. This is achieved via a recursive call to the lexer function, as illustrated in fig. 3.13.1.

```
** rules section of UFOLex
```

```
** ignore white space with a recursive call to yylex
[ \t\n]         =>  { return yylex }
```

Figure 3.13.1 UFOLex rule to discard white space.

Sometimes a token is of extra special interest to a parser. Such an example is when we scan a 'number'. Although the lexer function will return a token telling the parser that a number has just been scanned, the parser will also need to know it's value. This is known as the attribute of a token and is available to the action code through "`yytext`", which contains the string that was last matched. '`yytext`' is a convention adopted by most lexical analyser generators. Fig. 3.13.2 shows an example in which "`yytext`" is converted into an integer and returned to the parser. It also shows how an variable name or identifier in a programming language could be recognised and passed onto the parser.

```
** encapsulate an attribute value derived from "yytext" into
** "NUMBER" and return the token as the result of the yylex.
[0-9]+       =>  { value = read_int(yytext);
                   return NUMBER(value) }

** encapsulate attribute and return token as result.
[A-Za-z][A-Za-z0-9]*   =>   { return IDENTIFIER(yytext) }
```

Figure 3.13.2 Lexer return value encapsulation.

A compact and efficient representation is used to communicate this information from the lexer to the parser. This has to be largely under the control of the user because a parser could take any form, especially a hand written version. The solution is to provide a generic framework on which to base this communication activity upon. 'C' implementations of lexical analyser generators suffer from a clean solution because 'C' lacks polymorphism and generic types which would provide an excellent solution.

To pass this information to the parser, the approach is to generate a 'token' class and a series of subclasses to represent each type of token. Some classes, such as that for a number, would also have an encapsulated value. This would be assigned the token's attribute value before being returned from the lexer function to the parser function. This is possible due to the object oriented features of the UFO language. See fig. 3.13.3.

```
** defined by UFOLex
class Token end;
class EOF inherits Token end;
class ERROR inherits Token end;

** defined by parser, e.g. UFOYacc
class NUMBER(v : Int) inherits Token end;
class IDENTIFIER(n : String) inherits Token end;
```

Figure 3.13.3 Default and parser defined token classes.

UFOLex must define 'ERROR' and 'EOF' tokens itself because it must deal with these types of condition. Other tokens can be defined externally by hand or by an automatic parser generator such as UFOYacc.

3.14 Changing start state from within an action

This is achieved by calling a procedure '`yybegin()`' which will change the start state of the automaton, to the one which is specified. '`yybegin()`' is a standard used by most lexical analyser generators. Consider the example given in Fig. 3.14.1.

```
%%
%s COMMENT
%%
"**"                    =>  { yybegin(COMMENT); return yylex }
<COMMENT> \n            =>  { yybegin(INITIAL); return yylex }
<COMMENT> .*            =>  { return yylex }
```

Figure 3.14.1 example use of start states to strip comments from a UFO program.

Fig. 3.14.1 shows how a UFO comment could be recognised and discarded in a UFO compiler. A special state called 'COMMENT' is entered, for which only the second and third rules are valid. Having scanned the entire comment, we can return to normal compiling state 'INITIAL', having successfully discarded the comment.

3.15 Lexer conflicts

If two rules specified in the rules section conflict when given an input, the lexer resolves the

problem by choosing the longest match. When two rules match the same string exactly, then the rule that was listed first in the rules section is chosen. For example, in fig. 3.15.1, '`action1`' is executed on input "`hello`" even though `action2` could just as well be executed;

```
"hello"     =>          { action1; }
[a-z]*      =>          { action2; }
```

Figure 3.15.1 conflicting rules.

Patterns which use the trailing context operator, adopt a similar strategy. For example, given the two rules:

```
A/BCxxx
ABC/xxx
```

both rules will match the input 'ABCxxx' but the first rule is always declared as the matching rule. This may seem strange since the actual proportion of the string finally declared as having been matched is 'A' as opposed to 'ABC' with the second rule. However this makes sense because the actual amount of lookahead is greater in rule one than that in rule two and hence it must be the correct rule to declare as having matched the input.

3.16 Lexical Errors

There are few error conditions at a lexical analysis phase due to the analyser's limited view of the input string. However, if no token patterns match the prefix of the current remaining input string then the automaton simulator invokes a standard method called '`yyerror()`'. The default action of `yyerror()` is to report the message "`unmatched character`" and then terminate.

With a well designed set of rules, this type of error should be avoidable. To ensure this, a rule which consists of a '.' should be included for each possible start state in a UFOLex program. Reference [3] sec. 3.1 p. 88 gives a full discussion of strategies for error recovery.

3.17 Input from strings and buffers

Sometimes, lexical analysers need to look ahead a number a characters into the input string, in order to make a match and when finished, to push these characters back into the input string. This process can be greatly speeded up by using a buffer.

Most versions of Lex, use a buffer with a wrap around. This limits the maximum length of a recognisable lexeme to being the length of the buffer. ML-Lex, allows variable sized strings and arrays which eliminate this problem. UFOLex also adopts this approach because the functional aspects of the UFO language permit the treatment of arrays and strings in a similar way.

An application may wish to draw from several different input sources e.g. files, strings, keyboard etc. Similarly, it may wish to perform file nesting and file chaining. The method adopted by Flex is to implement a set of standard functions which create and maintain different types of buffer. In UFOLex, the user can achieve this by simply utilizing the object oriented nature of the UFO language. New 'buffer' objects can be created by writing methods that are specific to a particular input source. UFOLex thus provides a programmer friendly object-oriented framework for this activity[1]. Also, the ability to explicitly match an EOF character allows file chaining to be cleanly implemented.

3.18 Automatic line number counting

To assist with error reporting, the lexical analyser automatically keeps track of line numbers in the input by examining the current input buffer. This raises problems when dealing with nested files and when using file chaining. Thus, in a lexical analyser generated by UFOLex, each instance of a buffer contains it's own line number counter. The current line number is available to a user action at any time via the variable '`yylineno`'.

1. The default input source for a lexical analyser produced by UFOLex is stdin.

4. UFOYacc

4.1 Role of UFOYacc

UFOYacc automates the construction of syntax analysers. UFOYacc generates an efficient push down automaton recogniser for a given grammar specification, similar in notation to Backus-Naur Form[1], which can be used to parse a large class of context-free grammars. These are called LR(1) grammars. For a full description of LR parsing, refer to [3] sec. 4.7. Context-free grammars differ from regular grammars as used by UFOLex. For a description of the difference see [3] sec. 4.2. UFOYacc also incorporates an attribute grammar scheme with synthesised attributes. See [3] sec. 5.1 p.280-p.282.

A LR(k) parser gets its name from the way that it works. It reads input from left to right and produces a right-most derivation of the parse tree. It uses a lookahead of k tokens and works from the bottom up. This parsing method looks for tokens to complete a sentence. When a sentence has been gathered, the parse tree containing these tokens can be pruned. Repeated pruning occurs as more complex sentences are identified, until all that is left is the starting symbol of the grammar at the root of the parse tree. This indicates a complete and successful parse of the input.

A parser's main task is to check whether strings of tokens, generated by a lexical analyser, conform to well defined language constructs. The parser is required to execute actions triggered by the recognition of valid sentences within a language. The parsing algorithm constructed from the input grammar is often referred to as a '*shift/reduce*' parser.

4.2 Operation of UFOYacc

UFOYacc operates in a similar fashion to UFOLex, providing a mechanism for solving syntax analysis problems such as language recognition. It takes as input, a UFOYacc program, specified on the command line ('.y') and it generates a '.ufo' module containing the resulting parser. This can be readily combined with a lexical analyser generated by UFOLex and any other application program by using the UFO compiler to link everything together. See Appendices A.3 and A.4 for an example UFOYacc input program and it's resulting parser generated by UFOYacc.

1. BNF is introduced in [3] sec. 2.2

4.3 UFOYacc features

In order to construct correct parsers, UFOYacc provides mechanisms which help the process of constructing a correct parser much easier. These features are particularly useful as an aid when processing common computer programming language constructs, for example, the dangling else problem[1]. All mechanisms are activated via the UFOYacc input program. Below is an outline of some of the most important features:

1. Representing LR(1) grammars using Backus-Naur notation.
2. Grammar disambiguation.
3. Assigning precedence and associativity.
4. S-Attributed syntax directed translation and user action code.
5. S-Attribute variable types.
6. Terminal and Nonterminal types.

4.4 UFOYacc programs

A UFOYacc program specification draws from features in existing Yacc utilities such as 'byacc' and 'ML-Yacc'. The general organisation of a UFOYacc input program is identical to that of UFOLex (See Chapter 3, section 4). However the contents of the last two sections of the input program are different. The BNF rules section contains a series of grammar productions and with each production is an associated user action, written as a UFO program fragment. Other UFO code can be given in the user declarations section at the beginning of the UFOYacc input program, if it is required to be in the same scope as the user actions. In a similar way to UFOLex, all UFO code within the UFOYacc input program, is copied verbatim to the resulting parser.

Appendix C.1 gives the complete syntax for the UFOYacc input program including lexical conventions.

4.5 Grammar rules

Grammar rules are specified in a similar manner to a BNF production rules. They consists of a lefthand side and a righthand side, delimited by a colon and terminated with a semicolon. The lefthand side consists of a single nonterminal symbol and the righthand side consists of a list of terminals and/or nonterminals. Fig. 4.5.1 shows examples of two

1. See reference [3] sec. 4.3 p.174 - p.175

UFOYacc grammar rules, which recognise the language of natural numbers. Terminals are written in capitals and nonterminals are written in lowercase. '`DIGIT`' is a terminal which would typically be retrieved by a lexer working in cooperation with the parser.

```
number : DIGIT
       ;

number : DIGIT number
       ;
```

Figure 4.5.1 Example UFOYacc grammar productions

In fig. 4.5.1 each grammar rule had the same lefthand nonterminal. Any number of rules with common lefthand nonterminals can be grouped using the '|' operator. This operator is identical to the *alternation* operator in UFOLex. It should be noted, that this does not change the meaning of the rules that have been grouped, it is merely a convenient shorthand. Fig. 4.5.2 shows the grouping of rules from fig. 4.5.1. One could read the new rule like this: **"a *number* is a single *DIGIT* OR a DIGIT followed by another *number*"**.

```
number : DIGIT
       | DIGIT number
       ;
```

Figure 4.5.2 use of the 'alternation' operator.

User actions are specified in a *similar* way to actions in UFOLex. For each grammar rule and any of it's alternatives, a user action must be specified by placing the UFO code fragment on the required line. Fig. 4.5.3 gives a simple example.

```
list : ELEMENT                      { ** user action 1 }
     | list COMMA ELEMENT           { ** user action 2 }
     ;
```

Figure 4.5.3 UFOYacc rule showing user actions.

4.6 UFOYacc definitions section.

The second section of the UFOYacc input program can contain a number of different

commands. Only a few commands are optional because most are necessary to complete the definition of the resulting parser which UFOYacc creates.

4.7 Terminals and Nonterminals

The user must specify a list of all the terminals and nonterminals used by the grammar in the rules section. This is important because it represents an opportunity to give type information to produce a *type-secure S-attributed syntax directed translation system*. For more information on syntax directed translation, see [3] sec. 5.1 p.280-p.282. In addition, the identifiers in the grammar rules section can be checked to make sure that they are valid. Finally, if an automatic lexical analyser such as UFOLex is used in conjunction with a parser generated by UFOYacc, then the terminals are used as an agreed communication protocol for valid lexemes, between the lexer and the parser.

The format of the commands for specifying terminal and nonterminal names is shown by example in fig 4.7.1. Type information is given in angled brackets after the `%term` or `%nonterm` commands and is followed by the list of symbols which are required to inherit that type. Note that `%term` and `%nonterm` can be used any number of times, as would be the case when dealing with many different symbol types.

```
%term <Int> NUMBER
%term <Buckets> FRED
%nonterm <Int> expression list program
```

Figure 4.7.1 Example terminal/nonterminal definitions.

4.8 Grammar disambiguation in Shift/Reduce parsing

In order to understand the concepts of grammar disambiguation, which is so heavily relied upon by the compiler writer, it is important to be aware of how *shift/reduce* parsing takes place at an abstract level.

The underlying operation involves the use of a stack onto which grammar symbols are pushed and popped. The general idea is that the parser tries to form the complete right hand side of a grammar rule by successively retrieving tokens from the lexer and pushing them onto a stack. This is called "*shifting*". Once a match to a complete righthand side has been

found, all the stacked symbols are popped and replaced with the lefthand side symbol of the grammar rule. This is called "*reducing*", for obvious reasons.

Ambiguity arises when the the parser is presented with grammar rules which generate languages which are subsets of each other. In this situation, the shift/reduce parser could either perform a shift by pushing another symbol onto the stack, or a reduce operation. These grammars are said to be ambiguous and hence cannot be classified as LR(1).

It is possible to re-write most grammars to avoid this situation, i.e. to make them LR(1). However, this approach leads to over complicated grammar descriptions. UFOYacc provides a means of extending the number of grammars it can handle by providing mechanisms for resolving known disambiguities. The user can specify disambiguation rules in the `UFOYACC DEFINITIONS` section of the UFOYacc input program.

4.9 Assigning precedence and associativity.

UFOYacc provides a simple system of specifying precedence and associativity of grammar rules and terminals in an attempt to disambiguate non LR(1) grammars.

Consider the ambiguous grammar of fig. 4.9.1 which suffers from a *shift/reduce* conflict. Two possible derivations could be obtained from an input of "X+X+X", depending on which course of action the parser chooses to take.

```
expr : X
     | expr + expr
     ;
```

Figure 4.9.1 Grammar requiring disambiguation.

If the *reduce* is chosen to resolve the conflict then the result will be "(X+X)+X". If the shift is chosen then the result would be "X+(X+X)". By default, UFOYacc always chooses the *shift*, but the user can use the definitions section of the UFOYacc input program to control this behaviour. This is achieved through the use of precedence and associativity.

Terminal symbols involved in *shift/reduce* conflicts can be assigned a precedence. In addition, every corresponding rule can be assigned a precedence. This is done automatically

by UFOYacc by adopting the precedence of the rightmost terminal in the rule. If there is no such terminal, then UFOYacc will look to the user to supply a precedence explicitly for that rule by using the `%prec` command.

The *shift/reduce* conflict can be resolved by comparing the precedence of the terminal to be shifted, with the precedence of the rule to be reduced. It shifts if the terminal's precedence is greater and reduces if the rule's precedence is greater. However, this is often insufficient for solving all *shift/reduce* conflicts. For example, the grammar of fig. 4.9.1 would not benefit from such treatment because the precedence of the terminal symbol to be shifted is the same as the precedence of the rule to be reduced.

The solution is to turn to associativity to resolve the conflict. If the terminal to be shifted is left associative then the parser chooses to reduce. If the terminal is right associative, then the parser chooses to shift. If the terminal if non-associative then UFOYacc will generate an error. To assign associativities for such a test, UFOYacc provides three commands. `%left`, `%right` and `%nonassoc`

We can explicitly disambiguate the grammar of fig. 4.9.1, by using "`%left +`" to tell UFOYacc it should be generating "(X+X)+X" instead because the "+" operator is left associative.

Fig 4.9.2 shows a grammar combining the aspects described above whilst giving a clear picture of how to use the commands in the declarations section.

```
%%
%left MINUS
%left MUL
%right UMINUS
%%
expr : expr MINUS expr              { action1; }
     | expr MUL expr                { action2; }
     | MINUS expr %prec UMINUS      { action3; }
     | NUMBER                       { action4; }
     ;
```

Figure 4.9.2 A disambiguated grammar for simple arithmetic expressions.

`%left` is used to assign associativity to the subtraction and multiplication operators. `%right` is used to give associativity to the negation operator. The order in which the associativities are assigned reflects their precedence too, with the lowest precedence going to the first terminal to be specified and the highest precedence going to the last terminal to be specified. "`UMINUS`" has the highest precedence. It is a bogus terminal whose sole purpose is as a means of assigning a high precedence to a rule via the `%prec` mechanism. Hence negation will always occur before subtraction and multiplication, yet subtraction will still occur after multiplication. This is sufficient to resolve all *shift/reduce* conflicts for this grammar.

Another class of conflict is the *reduce/reduce* conflict. Fig. 4.9.3 shows an example of a grammar with such a conflict.

```
prog : proga
     | progb
     ;
proga : X
      ;
progb : X
      ;
```

Figure 4.9.3 Grammar exhibiting a reduce/reduce conflict.

In this example, the parser, having seen '`X`' doesn't know whether or not to reduce it to '`proga`' or '`progb`'. The strategy adopted by UFOYacc to resolve this conflict, is to the reduce the rule with the highest precedence. If rule precedences have not been explicitly specified by the user with the `%prec` command on the relevant rules, then UFOYacc's default behaviour is to reduce the rule listed first in the UFOYacc input program.

4.10 S-attributed syntax directed translation and user action code

As mentioned in section 4.7, UFOYacc supports syntax directed translation. This involves performing *semantic actions* using the context-free grammar descriptions as a guide. Hence they are evaluated bottom-up and left-right through the parse tree. The *semantic actions* often involve performing such tasks as type inference and intermediate code generation in compilers.

Semantic actions are embedded by the user into the UFO code fragments associated with each grammar production. Each grammar symbol in a given production is assigned an attribute value and a corresponding attribute type. This information is available to the UFO code fragments through local variables automatically created by the parser.

Fig. 4.10.1 shows an example grammar and its associated semantic actions. A simple local variable naming scheme is used to uniquely identify the attribute value of each grammar symbol on the righthand side of each production. In this scheme, *{symbol_name}{n}* gives the local variable name for the nth occurrence of *symbol_name* in the grammar rule, starting from the left and counting through to the right. A new attribute value for the lefthand side nonterminal can be synthesised and stored by the parser by simply returning the value as the result of the UFO code fragment. This is a simple description of 'S-attributed translation'.

```
expr : NUMBER           {return NUMBER1}
     | expr PLUS expr   {return expr1 + expr2}
     | expr MULT expr   {return expr1 * expr2}
     | LP expr RP       {return expr1}
     ;
```

Figure 4.10.1 Syntax directed translation of an arithmetic expression.

The arithmetic expression described by the grammar of fig. 4.10.1 is evaluated in the order in which it is parsed. It is vital that each code fragment returns an attribute value of the correct type. The user can define the type of individual nonterminals and terminals for this purpose as described in section 4.7 where mechanisms for defining terminal and nonterminal symbols were introduced.

Note that *empty productions,* i.e. grammar productions with no righthand side must still specify a UFO code fragment and must still return an attribute value of the correct type.

4.11 Start symbol

The user must specify a starting symbol for the grammar with the `%start` command in the definitions section of the UFOYacc input program. If this command is omitted then UFOYacc will use the nonterminal for the first listed rule as the starting symbol. Each successive parse returns the value associated with the start symbol.

5. UFOLex Design

5.1 Design overview

The task faced by UFOLex is to construct a recogniser that looks for specific lexemes in the input stream. The recogniser must exhibit the following behaviour. Within a given section of input;

1. the recogniser always matches the longest lexeme if it finds matches to several different patterns.

2. the recogniser will always declare the first listed pattern in the UFOLex program as having matched, if there are two or more patterns that match the same longest lexeme.

Since lexing involves examining each input character individually, it is often a slow task and can account for a significant amount of compilation time for a typical compiler. It is important to minimise the number of operations performed per input character, in order to maximise performance.

5.2 Lexical analyser breakdown

There are many aspects of the design of UFOLex which correspond to the key algorithm behind it. This algorithm has to take regular expressions and create a finite automaton, which in turn is represented by a finite automaton simulator and transition table, in the output created by UFOLex. This constitutes the recogniser.

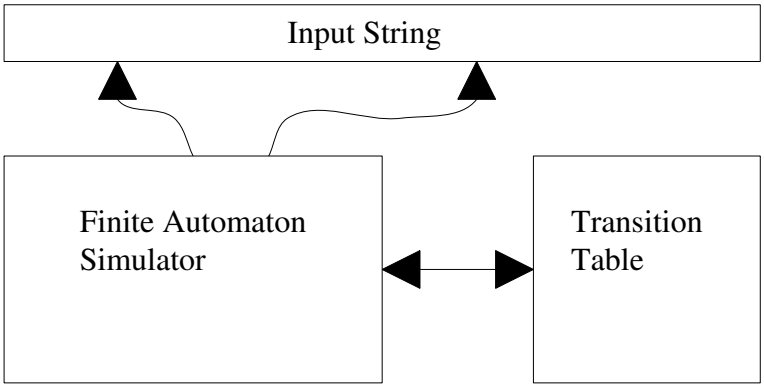


Figure 5.2.1 Lexical analyser overview

Fig. 5.2.1 shows a schematic of the components and operation of a lexical analyser generated by the UFOLex compiler. A lexeme is scanned from the input string by the finite automaton simulator which uses a transition table to lookup valid transitions. If a transition exists for that lexeme from the current state, then the internal state of the finite automaton simulator algorithm is changed.

5.3 Finite automaton simulator

A finite automaton (finite state acceptor) can be constructed for a grammar to recognise whether a sentence is in a language. The simulator does the following:

1. Reads input symbols one at a time
2. Changes its internal state on consumption of a valid symbol by consulting a table of transitions.
3. Terminates when either the input is rejected or when all input has been consumed. (i.e. EOF encountered in input)

In UFOLex, we also require that user actions are executed upon reaching special internal 'accepting states'.

The implementation of UFOLex could be based on either nondeterministic or deterministic automata[1]. For a full discussion see [3] sec. 3.6 p.113 - p.117. Since speed and size of the resulting simulator is important to us, the deterministic approach is chosen.

5.4 Choice of finite automata generator algorithms

There are many approaches to generating a deterministic finite automaton (DFA):

1. Construct an NFA from a regular expression using Thompson's construction. Then simulate the behaviour of the NFA by an algorithm that performs subset construction at run-time.
2. Construct an NFA from a regular expression using Thompson's construction and them use the subset algorithm to convert the NFA into a DFA. Then simulate the behaviour of the DFA.
3. Construct a DFA directly from a regular expression using a modified subset algorithm, then simulate the DFA.

1. NFA and DFA for short

All these approaches are syntax directed approaches in that they use the syntactic structure of regular expressions to guide the construction process. Approach three is the most appealing due to its simplicity. Other implementations of Lex, such as ML-Lex also choose this approach. The algorithm is based on the algorithm given in [3] section 3.9, p.134 - 141. Since it is syntax directed in nature, one important step in the design of UFOLex, is to create a scanner and parser for regular expressions. The parser will have to automatically generate a syntax tree from regular expressions.

5.5 Regular expressions

Regular expressions are a recursive notation which allow us to define any language derived from a regular grammar precisely in terms of a series of operations on that language. For more information about regular grammar properties see Appendix E.1. These operations form the node components used in the construction of the syntax trees required by the finite automaton generation algorithm. Parentheses are used to group subexpressions, however unnecessary parentheses can be omitted by observing the precedence order of these operators.

5.6 Language operations and regular expressions

Union (alternation), *concatenation* and *closure* are the three main operations which can be applied to languages represented by regular grammars. See [3] sec. 3.3 p.94 fig.3.8 for a more formal definition of each operation, where it should be noted that '*positive closure*' can be redefined in terms of '*Kleene closure*'.

The recursive structure of regular expressions using these operators can easily be described using a syntax tree gathered from a parser for regular expressions. Reference [3] sec. 3.3 p.94 - p.95 shows the syntax tree components as rules which define regular expressions over a vocabulary Σ.

5.7 Conversion of notational shorthands

Each shorthand provided to improve the usability of regular expressions can be re-written in a form consisting of only those the language operations described in section 5.6, which define classes of regular expression. Once a regular expression is re-written in this way, it is

suitable enough to be added to the syntax tree used by the finite automaton generator algorithm. For example consider r? which means zero or one occurrence of 'r'. It denotes the language L(r) ∪{ε}. In other words r? ≡ r | ε. For a full list of conversions, see Appendix B.3.

Context specifier notations are not quite so simple. For the specification of context within regular expressions, other regular expression rules/constituent parts must be defined.

1. The trailing context operator '/' is different from those of concatenation, alternation and closure defined above. It is easiest to treat '/' more like a special basic symbol in the syntax tree e.g. similar to that of ε. However we do not actually look for '/' on the input. When the pattern described by a regular expression containing a '/', is matched, the end of the lexeme is treated as being the position with the '/' transition, as opposed to the end of the overall regular expression. The '/'-transition forms one of the basic symbols which decorate a syntax tree at it's leaves.

2. The right context symbol '$' can be written in terms of the trailing context operator '/'. Given 'r$' we can re-write this as 'r / \n'. Normally the trailing context operator is used only once in a given regular expression. Hence given 'r$', r cannot include the trailing context operator.

3. The left context symbol '^' can not be re-written in a regular expression form. It's implementation requires the manipulation of flags within the buffers of the resulting lexical analyser.

The idea of regular definitions can be incorporated into the syntax tree by adding new branches to represent a particular regular definition whenever it's substitution occurs within a regular expression.

5.8 DFA construction algorithm

We have seen how the notation used by UFOLex to describe a regular expression can be broken down into a few basic components. The next step is to see how the finite automaton generation algorithm uses these components. See [3] sec 3.9, p.140 - p.141 The algorithm uses a method which groups subsets of states of an NFA[1] for a regular expression in an attempt to make the automaton deterministic. This is a modified version of the general

1. Nondeterministic Finite Automaton

subset construction algorithm given in [3] sec. 3.6, p.118 - p.121.

The key difference between this algorithm and the normal subset construction algorithm, is that is uses a syntax tree generated from parsing a regular expression, as opposed to a state transition diagram as an intermediate representation of it's NFA. For more information on state transition diagrams refer to [3] sec. 3.4 p.99 - p.104.

When the subset construction algorithm is used to turn an NFA into a DFA, one general idea is that, in an NFA transition table, each entry is a set of states. But in the transition table for a DFA, each entry is just a single state. Hence each DFA state corresponds to a set of NFA states. The other idea is that un-important states are ignored. A state is considered important if it has 'out' transitions that are non-null (ε). These are the only states used by the subset algorithm in the construction of a DFA transition table.

The nice property of a syntax tree generated from a regular expression, is that it is easy to distinguish these "important" states within it. Further more, positional relationships between leaves of the syntax tree, represent the same information as the ε-transitions in a state transition diagram, which provide a means of determining when one state can follow another.

To simplify this DFA construction algorithm by reducing the number of tests required when determining if a state is important, we can make use of the fact that there is only one accepting state in an NFA which is considered an un-important state. This is because it has no out transitions at all. We can make the accepting state important by concatenating a transition to it. This takes the form of a right hand marker (#) on the regular expression (r) e.g. (r)#. What results is called an *augmented regular expression*. It allows the accepting states to be ignored as the subset algorithm proceeds because now any state with a #-transition must be an accepting state.

5.9 Syntax tree constituent components

An augmented regular expression is represented by a syntax tree with the basic symbols involved in transitions positioned at the leaves and the operators for these symbols positioned at the interior nodes. The operators, as we saw earlier, consist of concatenation, alternation and kleene closure. The basic symbols include members of the ASCII character set, the # marker representing the end of a regular expression, the '/' marker representing

trailing context and the empty string ε.

A recursive datatype would be ideal for describing this tree structure. In UFO, a system of classes and inheritance can be utilised to provide the same thing. Below is a list of the objects that will go into the makeup of a syntax tree.

At the top level, we have an expression 'Exp' given by:

```
class Exp end;
```

At the leaves of the syntax tree, we have the most basic constituents of an expression, which consist of either an ASCII character:

```
class Chars(ascii : Char) inherits Exp end;
```

an ε character:

```
class Empty inherits Exp end;
```

an end of regular expression marker (#):

```
class End inherits Exp end;
```

or a trailing context marker:

```
class Trail inherits Exp end;
```

At the interior nodes of the syntax tree, we have either the closure operator of some expression x:

```
class Closure(x : Exp) inherits Exp end;
```

the alternation operator of two expressions x and y:

```
class Alt(x, y : Exp) inherits Exp end;
```

or the concatenation operator of two expressions x and y:

```
class Cat(x, y : Exp) inherits Exp end;
```

For example, given the regular expression (a|b)*/bc we can construct the syntax tree given in fig. 5.9.1.

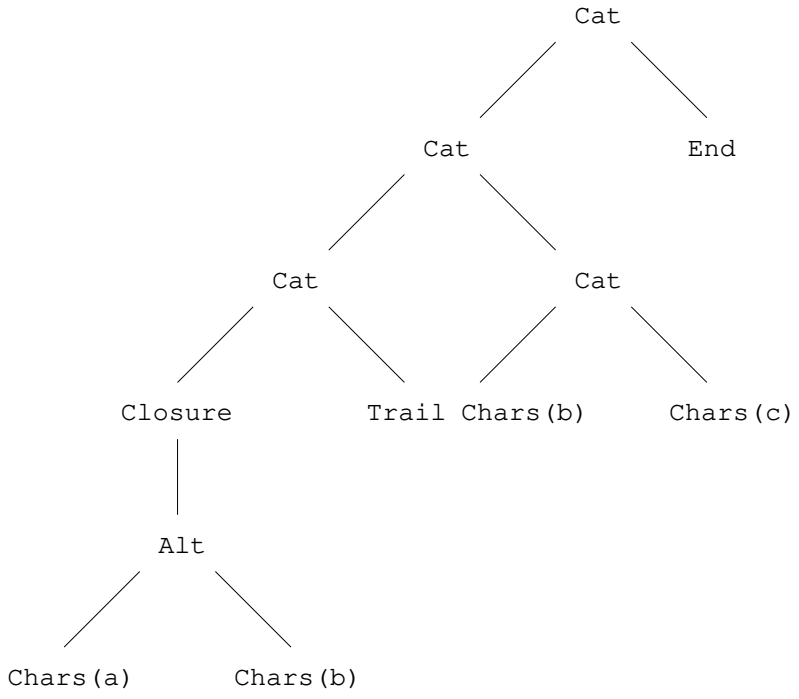


Figure 5.9.1 Syntax tree of (a|b)*/bc

The syntax tree of fig. 5.9.1 is assembled in UFO using:

```
Cat(Cat(Cat(Closure(Alt(Chars(a),Chars(b))),trail),Cat(Chars(b),Chars(c))),End)
```

and is almost identical to the parse tree obtained for this regular expression, but this is not always the case. The parse trees of other regular expressions need altering to become syntax trees by re-writing such things as notational shorthands and regular definitions, as described earlier.

5.10 Decoration of syntax tree with position information

To each leaf containing a basic symbol, other than the ε symbol, we attach a unique integer value. This represents the position in the tree of a particular basic symbol. Note that if a basic symbol occurs more than once in the syntax tree, then that symbol will be assigned

several position values, one for each position at which it occurs. These correspond to the numbered states of the equivalent NFA state transition diagram. Below is a final modified version of the above syntax tree class structure incorporating position numbering.

```
class Exp end;
class Empty inherits Exp end;
class Chars(pos : Int, ascii : Char) inherits Exp end;
class End(pos : Int) inherits Exp end;
class Trail(pos : Int) inherits Exp end;
class Closure(x : Exp) inherits Exp end;
class Alt(x, y : Exp) inherits Exp end;
class Cat(x, y : Exp) inherits Exp end;
```

We walk the tree from left to right and from bottom to top. Each non ε leaf that we encounter on our walk, is given an integer value which is then incremented for the next leaf we find.

One aspect of the subset algorithm mentioned earlier was that only "important transitions" are considered. By assigning a position numbering to the leaves of the syntax tree in this manner, we yield a tree which has only it's "important" transitions numbered. Now we are in a position to build a DFA from the syntax tree by creating states from sets of positions in the tree. But first we shall see how to parse the UFOLex source program, because it is the parsing process which allows us to easily generate these syntax trees.

5.11 Scanning and parsing a UFOLex program

First we need a grammar specifying the exact syntax of a UFOLex program. Appendix B.2 gave an EBNF description of this syntax. Appendix B.4 shows the equivalent BNF description. In a BNF form it is easier to identify the work done by the parser and the work done by a lexer. Terminals are handled by a lexer and nonterminals are dealt with by the parser.

Each non-terminal of the UFOLex program grammar description can be converted into an individual function. These functions constitute a 'recursive decent parser'. This is the simplest and most error free method of generating a parser by hand. However, the only restriction imposed by the recursive decent method, is that we must eliminate left recursion from our grammar and perform left factoring to eliminate backtracking. Appendix B.5 shows the resulting grammar after having performed these operations.

Having achieved it's goal, the parsing process has produced syntax trees for augmented regular expressions, translating any notation shorthands into operators used by the finite automaton construction algorithm. It has also gathered information relating to start states, and user code. Errors in syntax are reported and hence a check that the UFOLex program is correctly specified has been carried out.

5.12 Constructing the DFA

DFA construction can be viewed as having three complex steps:

1. Build a syntax tree for (r)# as described above.
2. Compute `nullable()`, `firstpos()`, `lastpos()` and `followpos()` by traversing the syntax tree.
3. The DFA is constructed from `followpos()` using the subset construction.

The process of DFA state construction involves looking at all possible transitions out of a DFA state. This may present us with the choice of entering several NFA states. A new DFA state is simply a grouping of all these NFA states. In effect, we short-circuit the production of an NFA by building a DFA whose states correspond to sets of positions in the syntax tree. `followpos()` is used to calculate which out transitions are valid.

We define `followpos()` on every position in the syntax tree, to obtain position matching of symbols in the input stream. To compute `followpos()` we need to know the positions within a node's subexpression of all possible first and last symbols. This information is created by `firstpos()` and `lastpos()`. These functions also require `nullable()` to complete their definitions. Each function is computed by walking the syntax tree from bottom to top and from left to right.

5.13 Subset and transition table construction

To construct the states of the DFA we use the `followpos()` information derived earlier. Using this form of the subset algorithm, we construct the deterministic finite automaton D = <T, Q, P, qo, F> using the subset algorithm to give:

T is a finite set of terminal symbols

Q is a finite set of sets of states where each set corresponds to a single state in the DFA
P is a finite set of deterministic productions of the form;

ql -> q'

where q, q' ∈ Q and l ∈ T
q0 is a start state of the DFA where q0 ∈ Q
F is a set of DFA accepting states where F ⊆ Q.

Initially, the start state 'q0' is `firstpos()` of the root of the regular expression. The algorithm is then given by reference [3] sec. 3.9 p.141 fig. 3.44.

In the subset construction algorithm P can be considered as a transition table '`tran`', where `tran[q,l] = q'` is equivalent to the production rule ql -> q'.

5.14 Incorporating multiple regular expressions

The general algorithm given by [3] sec.3.9 p.141, only deals with a single regular expression, producing a single transition table. We require multiple regular expressions, but still producing a single transition table.

To handle this behaviour we need to create a transition table containing all the transitions of every regular expression. Instead of the initial state q0, for the DFA construction algorithm being the `firstpos()` of a single regular expression, it is now the union of the initial states of all the regular expressions. This is equivalent to constructing a composite pattern of all the regular expressions e.g. e1 | e2 | ... | en. The result still fills a single transition table.

Thus, as a pre-requisite, we will be required to calculate the `followpos()` function for all the regular expressions. Also, when numbering the important leaves of each syntax tree, we must make sure the position numbers remain unique from one syntax tree to the next, not just within a single syntax tree. This allows us to exploit the fact that a pattern's accepting position can be used to uniquely identify it, for both announcing a matching pattern and for choosing which user action to execute as a consequence.

Once the subset construction algorithm has been executed, the next job is to create a DFA automaton simulator. It must exhibit a behaviour in which, if more then one pattern matches the input string, i.e. when one pattern is the prefix of another, the recogniser is to choose the longest lexeme matched. If there are two or more patterns that match the longest lexeme, the

first listed pattern in the UFOLex program is chosen. A simplistic algorithm for this is given in [3] sec. 3.6, p.116, Algorithm 3.1.

One key feature of the subset construction algorithm is that it yields information which makes the task of these decisions much easier. It allows them to be determined at compile time as opposed to when the DFA simulator is in operation. The subset construction algorithm produces a set of DFA states Q, which consist of subsets of syntax tree positions. If one of those positions yields an 'End' symbol in the syntax tree, i.e. a DFA state in Q contains an accepting position, then we know that when the DFA simulator enters this state, it will have matched a pattern.

The DFA simulator operates on multiple patterns by extending it to continue making state transitions even after one or more DFA accepting states are encountered. However, each time a DFA accepting state is encountered, the simulator records the pattern declared as having been matched and the position in the input string. Then it continues looking for more transitions. When a transition to a DFA state which comprises of more than one accepting position occurs, then this means that two patterns have matched a lexeme of the same length. UFOLex chooses to record the pattern which occurs first in the UFOLex program. This is simple to determine, because if we assigned position values to each syntax tree in the order that they appeared in the UFOLex program, then the first occurring pattern will have a smaller accepting position value than the other pattern. Hence a simple comparison can be used to choose which pattern to record.

Termination occurs when no further state transitions can take place for the given input symbol. At this point, we declare that the most recently recorded pattern is the one that matches the longest lexeme for the given input. We will then have to retract the current position of the input string to the position just after the longest matching lexeme. Luckily, we recorded this position when we recorded which pattern matched the input string.

As mentioned earlier, these decisions can be made at compile time. In the automaton simulator, we don't know if a DFA state is an accepting state because we can't look within the state to see whether or not it contains any accepting positions. The solution is to generate an extra table for use by the lexical analyser. It could be indexed by DFA state number and could contain accepting position values. Those DFA states that are accepting states would have an accepting position at their index. Those that are ordinary DFA states would not. Using a single lookup operation, the DFA simulator could determine which pattern to declare

as having matched the input string, if any, when in a given DFA state. It can then be used again later to execute the appropriate user action.

The subset construction algorithm given in [3] sec. 3.9, p. 141. can be modified to generate the decisions about which pattern to announce as having matched a lexeme. It uses the function `get_earliest()` to calculate which pattern should be recorded. It then records this in the '`pattern`' table, which is indexed by the DFA state number. To simplify the task of recording the pattern, we can just store it's accepting position. Fig. 5.14.1 shows pseudo code of the necessary additions to the algorithm.

```
...
while there is an unmarked ...
    ...
    if U contains any accepting positions then
       ** choose which to declare as the matching pattern.
       pattern[U] = get_earliest (accepting positions in U);
    else
       ** non accepting states don't match any patterns.
       pattern[U] = 0;
    fi
    ...
end
```

Figure 5.14.1 Modifications to DFA construction algorithm

5.15 Simulator enhancements

Fig. 5.15.1 shows the enhancements made to the DFA simulator over the original algorithm given in [3] sec. 3.6. p.116. It uses the '`pattern`' table and the transition table generated from the modified subset construction algorithm.

```
s := q0;
c := nextchar;
match := 0;              ** currently no pattern has matched
bufferpos := forward;    ** because no input has been read.
while c « eof do
   s := move(s, c);
   c := nextchar;
   if pattern[s] « 0 then    ** s must be an accepting state so,
```

```
            match := pattern[s];   ** 1. record pattern number
            bufferpos := forward;  ** 2. record buffer pos
         fi;
         if s = 0 then           ** if no more transitions could be made
                                 ** then we have matched a pattern.
            forward = bufferpos; ** retract buffers forward pointer
            return "matching pattern: " ++ match.print
         fi
      endwhile;
```

Figure 5.15.1 Modified DFA simulator

The '`pattern`' table is also used to choose the user action to be executed when a pattern matches an input string. Most versions of Lex generate a case statement into which the user actions are pasted. UFOLex can adopt the same strategy.

5.16 Incorporating start states

The design of start states into UFOLex involves further changes to the DFA generation algorithm. In the previous description of the algorithm, all regular expressions were combined into a single regular expression through alternation operators. The starting state was subsequently calculated from '`firstpos(root1 | root2 | .. | rootn)`', where `root1, root2 .. rootn` are the syntax tree roots for each regular expression.

Exclusive start states can be thought of as mechanisms for cutting down the number of regular expressions combined by the alternation operator. By combining different start states together and excluding others, we can calculate several different starting states for the DFA simulator. If the DFA simulator is given one of these 'start states', only the subset of regular expressions combined through alternation, will try to be matched. In a UFOLex program, we can associate a label to describe which regular expressions will be matched when in a given start state. If no labels are given in the UFOLex program, then all syntax trees are combined together. Any regular expressions that are specified without a label are grouped together to form the very first start state. By default, we call this state 'INITIAL'. There must be at least one regular expression defined in this way, otherwise we can't sensibly start the DFA simulator.

5.17 Trailing Context

A complicated addition to the DFA simulator algorithm, allows us to incorporate regular expressions utilising trailing context. The extra work required closely relates to the method of matching the longest lexeme.

When the automaton simulator matches the longest lexeme, we must retract the forward pointer by two amounts. The first amount is back to the end of the lexeme and if the pattern uses trailing context, then a second amount is used to cause the forward pointer to be retracted back to discard the trailing proportion of the lexeme.

During the simulation, we must record a forward pointer position for every single trailing context state encountered. This is because we don't know until the end of 'yylex', which pattern is finally chosen to match. A trailing context state is a DFA state consisting of one or more syntax tree positions labelled with 'trail', i.e. the trailing context operator.

We need to store the forward pointer positions so that they can be retrieved easily. The first thought is to associate each recorded forward pointer position with it's rule. This is fine for most cases, but fails when a DFA state consists of more than one trailing context position. The DFA simulator would be unaware of this. As a result, it would have no way of knowing which trailing context position and hence which forward pointer position belonged to which rule. A simple association of forward pointer position and rule number would not be possible.

The solution is to incorporate an extra level of indirection, by firstly associating each recorded forward pointer position with it's corresponding trailing context state. Then to associate each of these trailing context states with their rules. This gives rise to a one to many relationship which could be implemented using two simple lookup arrays. One array stores the forward pointers and the other array acts as a method of indirectly accessing these pointers again later.

In order to record forward pointers, the DFA simulator must be able to make the distinction between an ordinary state and a trailing context state i.e. the subset of syntax tree positions constituting the state, consists of one or more positions labelled with '`Trail`'. We can exploit the fact that a trailing context position uniquely identifies it's regular expression, because a

regular expression can only have one trailing context operator and every syntax tree position is unique. A table of trailing context DFA states could be constructed by UFOLex and added to the automaton simulator. It would be indexed by state number and would contain flags to indicate whether or not a particular state was a trailing context state.

So far we have incorporated three extra lookup tables into the automaton simulator. The pseudo code below shows these additions.

```
...
** execute this for each new state entered during the simulation
if is_a_trailing_context_state[state] then
   forward_pointers[state] = forward;
fi
...
```

Each new state entered, '`state`', is checked to see whether or not it is a trailing context state. This is achieved by consulting '`is_a_trailing_context_state`'. If required, we can then record the forward pointer position in the '`forward_pointers`' table. Below is pseudo code for when the automaton simulator has found a match to a rule.

```
if trailing_context_rules[match] != 0 then
   trail_state = trailing_context_rules[match];
   forward = forward_pointers[trail_state];
else
   forward = buffer_pos;
fi
```

When the simulation is complete, the automaton simulator would check to see if the matching rule '`match`', uses trailing context. This is done by looking in the '`trailing_context_rules`' table. This yields the state number of a trailing context state, which can then be used to index the '`forward_pointers`' table to retrieve the forward pointer position. If the '`trailing_context_rules`' table doesn't yield a trailing context state, then the position of the end of the longest matching lexeme is used. We can then retract the buffer position back by the correct amount and retrieve the matching lexeme.

6 UFOYacc design

6.1 The LR parsing algorithm

UFOYacc can be broken down into four key phases. Parsing a UFOYacc input program, generating the canonical collection of LR(1) items, building the action and goto tables and constructing the type-secure S-attributed syntax directed translation scheme.

The general organisation of an LR(1) parser is illustrated in Fig. 6.1.1. The parser scans the input buffer and uses the action and goto tables in cooperation with a stack to guide the parsing process.

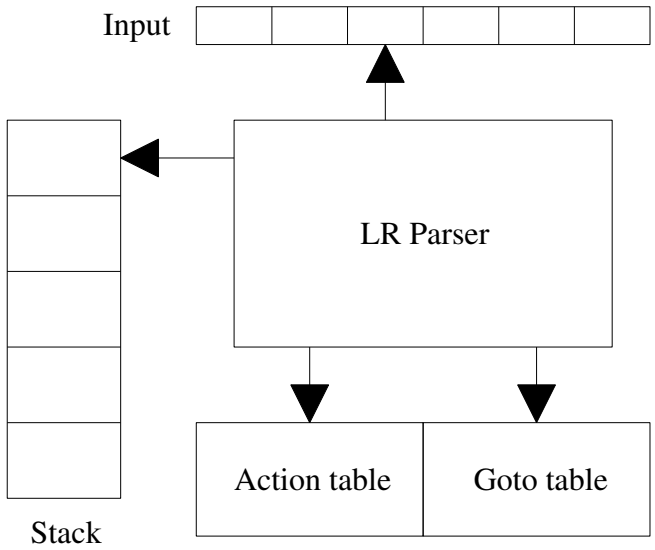


Figure 6.1.1 Schematic organisation of an LR parser.

It should be noted that the parsing algorithm and the organisation of the key components will allow SLR(1), LR(0), LALR(1), and LR(1) grammars to be parsed, the only difference being the contents of the action and goto tables.

See reference [3] sec. 4.7, p.216 - p.220 for a complete description of the algorithm behind the LR parser of fig. 6.1.1.

6.2 Parsing of a UFOYacc program

The design of UFOYacc with respect to parsing it's input programs, follows on from the

design of UFOLex. A recursive decent parser can be constructed, which helps identify the lexing tasks required by a separate lexical analyser. Appendix C.2 gives a version of the general UFOYacc program grammar of Appendix C.1. It has been left factored and left recursion has been eliminated so that it is suitable for the recursive decent method.

6.3 Construction of canonical LR items

The algorithm for constructing sets of LR(1) items is complex to say the least. Having implemented it, much work still remains to make the construction efficient. This project did not concentrate on efficiency because that is the drive behind using LALR(1) grammars. Reference [3] sec. 4.7, p.230 - p.234 gives a good description of the algorithm.

6.4 Construction of Action and Goto tables

[3] sec 4.7, p.234 - p.236, Algorithm 4.10 is used by UFOYacc to construct the action and goto tables of the resulting parser. It is at this stage that any *shift/reduce* or *reduce/reduce* errors are identified. They result when a new action table entry conflicts with an entry already present in the table. As an extension to algorithm 4.10 given in [3] sec 4.7, we can add a set of heuristics to help resolve these conflicts. Fig. 6.4.1 gives the pseudo code required to resolve shift/reduce conflicts. The functions '`precedence()`' and '`associativity()`' retrieve the precedence and associativity for a particular terminal or grammar rule as specified by the user in the UFOYacc input program. If the user has not specified any rules to disambiguate the grammar, then UFOYacc always to choose the shift.

```
on shift/reduce conflict:
   if no disambiguation rule for terminal is available then
      choose_shift
   else
      if precedence(terminal) > precedence(grammar rule) then
         choose_reduce
      elseif precedence(terminal) < precedence(grammar rule) then
         choose_reduce
      else
         ** same precedence
         if associativity(terminal) == Left then
            choose_reduce
         elseif associativity(terminal) == Right then
            choose_shift
```

```
            else
               error "shift/reduce conflict!"
            fi
         fi
      fi
```

Figure 6.4.1 Shift/reduce extensions to parsing table construction algorithm.

We can add more rules to help resolve *reduce/reduce* conflicts. Fig. 6.4.2 shows the necessary extensions to the basic algorithm. Again if no help in resolving disambiguities is supplied by the user, then UFOYacc always chooses to reduce the rule out of the two conflicting rules, which occurs first in the UFOYacc input program.

```
on reduce/reduce conflict:
   if precedence(grammar rule1) > precedence(grammar rule2) then
      reduce grammar rule1
   elseif precedence(grammar rule1) < precedence(grammar rule2) then
      reduce grammar rule2
   else
      reduce first listed rule in UFOYacc input program
   fi
```

Figure 6.4.2 Reduce/reduce extensions to parsing table construction algorithm.

6.5 Executing S-attributed action code

Since UFOYacc is in charge of generating a token for each Terminal and nonterminal symbol in a UFOYacc input program, it is able to attach attribute values to each one of them of the correct type. See Appendix A.4 for an example of this. These tokens are placed on the stack and are used to record any synthesised attribute values as the parsing progresses.

When a grammar rule has been matched, UFOYacc pops each grammar symbol off the stack and places their attributes in local variables so that they are available to the required UFO code fragment associated with the matching rule. Again, see Appendix A.4 for an example of this. It all takes place in a function called `'execute_production_rule()'`

7. Conclusions

7.1 Achievements

Both UFOLex and UFOYacc are completed utilities. There are only a few features that have been left out due to project time scale. They provide enough functionality to begin considering their use in aiding the implementation of the next version of the UFO programming language.

7.2 UFO as an implementation language

I have nothing but praise for UFO1.0/S. Other languages now seem very restrictive and counter intuitive to program in comparison. Of course, UFO1.0/S wasn't without it's problems. Appendix F.1 is a compilation of most of the problems that were encountered during implementation.

7.3 UFO language considerations

The original proposal to use ML-Lex and ML-Yacc as a basis for UFOLex and UFOYacc turned out to be difficult. UFO, after all, is a parallel programming language. Simple algorithms used by ML-Lex and ML-Yacc proved impossible to code directly into UFOLex and UFOYacc because they resulted in deadlocks.

The use of pointer types, i.e. references in ML, doesn't easily translate into UFO where the closest thing is a stateful object. It is undesirable to repeatedly use state in UFO. The functional aspects of the language are preferred.

Further more, the complexity of the algorithms behind Lex and Yacc are such that implementations are even more complex. It was thus necessary to go back to first principles and only use ML-Lex and ML-Yacc as rough guides for the direction the project should take.

7.4 Unimplemented aspects

Full error recovery has not been implemented in either UFOLex or UFOYacc. This is a large topic beyond the restrictive time scale allocated for a project of this size.

UFOYacc only creates parsers for LR(1) grammars. Most versions of Yacc implement

LALR(1) parsers. The difference is the efficiency of the resulting parser. Ironically, LR(1) can deal with more grammars than LALR(1) since LALR(1) is a subset of LR(1).

The efficiency of UFOYacc in constructing LR(1) parsers needs improving. The documentation provided with ML-Yacc, see [4], contains a "*Hackers guide to ML-Yacc itself*" in a file called "*tech.doc*". It provides a discussion of the "hacks" required in order to improve performance. Most compiler reference books omit this type of information.

In both UFOLex and UFOyacc, table compression algorithms were not implemented to improve the space taken to represent large grammars. Of course, this is not a theoretical problem, but would be desirable in a compiler. Reference [3] describes some common methods.

In UFOLex, the left context operator was left unimplemented and in both UFOLex and UFOYacc the ability to rename all identifiers in the resulting parsers and lexers was omitted. This would have allowed multiple parsers and lexers to be contained within a single UFO application without name clashes. In addition, a more structured class system in the resulting parsers and lexers would allow different grammars to be plugged in as attributes of general automaton simulator classes.

7.5 The next step - a UFO compiler.

Having built on UFOLex and UFOYacc to incorporate the features described in section 7.4, the next step would be to use UFOLex and UFOYacc to implement the next version of the UFO compiler. Due to the size of the UFO language, large array initialisations in the resulting parser will cause problems with the current UFO1.0/S compiler. A work around is simple but vital if the implementation of a UFO compiler is to be feasible.

References

[1] J. Sargeant, S.J. Hooton, I. Watson and C.Kirkham, "United Functions and Objects: an Overview"

[2] http://www.cs.man.ac.uk/arch/projects/ufo.html

[3] A.V. Aho, R. Sethi and J.D. Ullman, "Compilers: Principles, Techniques and Tools", Addison-Wesley, 1986.

[4] D. R. Tarditi, A and W. Appel, "ML-Yacc version 2.1", 1991.

[5] J. R. Levine, T. Mason and D. Brown, "Unix Programming Tools - Lex & Yacc".

[6] W. M. Waite and G. Goos, "Compiler Construction", Springer-Verlag, 1984.

[7] F. W. Weingarten, "Translation of Computer Languages", Holden-Day Inc., 1973.

Appendix A

A.1 Example UFOLex input program

```
** calc.lex
** lexer for example calculator
** (c) Mark Lomas

main : String is
{
   automaton = Automaton(LexBuffer(Stdio));
   calculator = Parser(automaton);
   return
      if calculator.yyparse then
         "O.K.\n"
      else
         "Failed\n"
      fi
}

%%

digit = [0-9];
number = {digit}+ ;
ws = [ \t];

%%
{number}      => { (value, rest) = yytext.read_int; return Number(value) }
\+            => { return Plus('+') }
\*            => { return Mult('*') }
\-            => { return Minus('-') }
\/            => { return Divide('/') }
\^            => { return Carot('^') }
\(            => { return Leftp('(') }
\)            => { return Rightp(')') }
\n            => { return Newline('\n') }
{ws}+         => { return self.yylex }
.             => { return self.yyerror("unexpected character") }
```

A.2 Example lexer generated by UFOLex from A.1

```
** calc_re.ufo
** DFA Automaton Simulator
** Generated by UFOLEX 1.0


class Startstate(s : Int) end
class yytoken end
class EOF inherits yytoken end
class Error inherits yytoken end

abstract class Input

   ** retrieve next character from input source
   getch : Char is deferred

end
```

```
** class to provide buffering of the input
stateful class LexBuffer(input : Input)
{
   state
      forward = 0;
      beginning = 0;
      line_no = 0;
      buffer = "";
}

   proc get_terminal : Int is
   do
      new buffer = if forward == buffer.length then
                      buffer ++ String(input.getch)
                   else
                      buffer
                   fi;
      result = new buffer@forward;
      new forward = forward + 1;
      return
         if Int(result) == -1 then 255 else Int(result) fi
   od

   line_number : Int is
      line_no

   lexeme_forward : Int is
      forward

   lexeme_beginning : Int is
      beginning

   proc update_buffer(new_line_no, new_forward : Int) : Void is
   do
      new line_no = new_line_no;
      new beginning = new_forward;
      new forward = new_forward;
   od

   private update_lineno(lexeme : String; count : Int) : Int is
      if lexeme == "" then
         count
      else
         if lexeme.head == '\n' then
            self.update_lineno(lexeme.tail, count+1)
         else
            self.update_lineno(lexeme.tail, count)
         fi
      fi

   get_lexeme(position : Int) : String is
   {
      lexeme = buffer.substr(beginning+1, position);
      self!update_buffer(line_no + self.update_lineno(lexeme, 0), position);
      return
         lexeme
   }
```

```
end


** Default input source is the Stdio!
** os!getchar will only return if a 'newline' or a CTRL-D character has been seen on the input.
** Thus generally, stdin is only interactive on a line by line basis.
class Stdio inherits Input

   ** os!getchar translates to the C call 'getchar' which returns -1 for EOF
   ** but UFO1.0/S just type casts this to a Char, hence EOF
   ** is represented by ASCII character code 255.
   getch : Char is
      os!getchar

end


** User Code
** calculator.lex
** lexer for example calculator
** (c) Mark Lomas


main : String is
{
   automaton = Automaton(LexBuffer(Stdio));
   calculator = Parser(automaton);
   return
      if calculator.yyparse then
         "O.K.\n"
      else
         "Failed\n"
      fi
}



update_table(array : Array_1[Int]; index, val : Int) : Array_1[Int] is
   (for i in [0 to 255] do
      new_entry = if i == index then
                     val
                  else
                     array[i]
                  fi;
    return all
       new_entry od).set_lower_1(0)

initialise_table : Array_1[Int] is
{
   row1 = [0..255 | 2,
2,2,2,2,2,2,2,2,11,12,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,11,2,2,2,2,2,2,2,10,9,8,7,2,6,2,5,4,4,4,4,4,
4,4,4,4,4,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,3,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0];
```

```
   row2 = [0..255 | fill 0].set_lower_1(0);
   row3 = [0..255 | fill 0].set_lower_1(0);
   row4 = [0..255 | 0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,14,14,14,14,14,
14,14,14,14,14,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0
```

```
      (yytext, yylineno, match) = self.scan(yystartstate, buffer.lexeme_beginning, 0, [1..15 |
fill 0]);

      ** might be nice to do
      ** yylex : yytoken is self.yylex

      return
         case match of
            0: null:EOF;
            1: { (value, rest) = yytext.read_int; return Number(value) };
            2: { return Plus('+') };
            3: { return Mult('*') };
            4: { return Minus('-') };
            5: { return Divide('/') };
            6: { return Carot('^') };
            7: { return Leftp('(') };
            8: { return Rightp(')') };
            9: { return Newline('\n') };
            10: { return self.yylex };
            11: { return self.yyerror("unexpected character") };
         otherwise:
            null:yytoken
         esac
   }

   scan(s, match_pos, match : Int; trail_store : Array_1[Int]) : (String, Int, Int) is
   {
      c = buffer!get_terminal;
      t = transition_table[(s-1)*256+c];
      (new_match_pos, new_match, go_on, new_trail_store) =
         if t==0 then
            if c==255 then
               (match_pos, match, false, trail_store)
            else
               if match_pos == buffer.lexeme_beginning then
               {
                  os!stderr("Unmatched Input.\n");
                  return
                     error : (Int, Int, Bool, Array_1[Int])
               }
               else
                  (match_pos, match, false, trail_store)
               fi
            fi
         else
         {
            ** if the new state is a trailing context state (i.e. trail_context_table[t]
            ** gives a corresponding accepting state) then record current lexeme forward
            ** pointer for use later if pattern is declared as matching.
            ntrail_store = if trail_context_table[t] != 0 then
                              update(trail_store, t, buffer.lexeme_forward)
                           else
                              trail_store
                           fi;
            return
               if pattern_table[t] != 0 then
                  if continue_table[t] == 0 then
                     (buffer.lexeme_forward, t, false, ntrail_store)
                  else
                     (buffer.lexeme_forward, t, true, ntrail_store)
```

```
                  fi
               else
                  (match_pos, match, true, ntrail_store)
               fi
         }
         fi;

      return
         if go_on then
            self.scan(t, new_match_pos, new_match, new_trail_store)
         else
            if new_match == 0 then
               (buffer.get_lexeme(new_match_pos), buffer.line_number, 0)
            else
            {

               altered_match_pos = if end_context_table[new_match] != 0 then
                                      new_trail_store[end_context_table[new_match]]
                                   else
                                      new_match_pos
                                   fi;
               return
                  (buffer.get_lexeme(altered_match_pos), buffer.line_number,
pattern_table[new_match])
            }
            fi
         fi
   }

   proc yybegin (s : Startstate) : Void is
   do
      new yystartstate = s.s;
   od

   yyerror(message : String) : yytoken is
   {
      os!stderr(message ++ "\n");
      return
         error : yytoken
   }

end
```

A.3 Example UFOYacc input program

```
** calc_grm.y
** example calculator
** (c) Mark Lomas 1997

expo(x, y : Int) : Int is
   if y == 0 then
      1
   else
      x * expo(x, y-1)
   fi

%%

%term <Char> Newline Plus Mult Leftp Rightp Minus Divide Carot
```

```
%term <Int> Number UMINUS
%nonterm <Bool> list printline
%nonterm <Int> expr

%left Minus Plus
%left Mult Divide
%right Carot
%right UMINUS

%start list

%%


list :                          {return true }
     | printline Newline list   {return true }
     ;

printline : expr                {os!message("= " ++ expr1.print ++ "\n"); return true}
          ;

expr : Number                   {return Number1}
     | expr Plus expr           {return expr1 + expr2}
     | expr Mult expr           {return expr1 * expr2}
     | expr Minus expr          {return expr1 - expr2}
     | expr Divide expr         {return Int(expr1 / expr2)}
     | expr Carot expr          {return expo(expr1, expr2)}
     | Leftp expr Rightp        {return expr1}
     | Minus expr %prec UMINUS {return -expr1}
    ;
```

A.4 Example parser created by UFOYacc from A.3

```
** calc_grm.ufo
** Generated by UFOYacc1.0

class Newline(yylval : Char) inherits yytoken end
class Plus(yylval : Char) inherits yytoken end
class Mult(yylval : Char) inherits yytoken end
class Leftp(yylval : Char) inherits yytoken end
class Rightp(yylval : Char) inherits yytoken end
class Minus(yylval : Char) inherits yytoken end
class Divide(yylval : Char) inherits yytoken end
class Carot(yylval : Char) inherits yytoken end
class Number(yylval : Int) inherits yytoken end
class UMINUS(yylval : Int) inherits yytoken end
class list(yylval : Bool) inherits yytoken end
class printline(yylval : Bool) inherits yytoken end
class expr(yylval : Int) inherits yytoken end

abstract class Action end
class Shift(i : Int) inherits Action end     ** Shift state i onto stack
class Reduce(j : Int) inherits Action end    ** Reduce by production numbered j
class Accept inherits Action end             ** Accept the current input
class Perror inherits Action end             ** used to fill in table blanks

** stack element = (state number, yytoken) pair, where yytoken is used to
** obtain values for synthesised attributes.
synonym Stack = List[(yytoken, Int)];
```

```
** Start of user declarations Section
** calculator.y
** example calculator
** (c) Mark Lomas 1997

expo(x, y : Int) : Int is
   if y == 0 then
      1
   else
      x * expo(x, y-1)
   fi

** End of user declarations section
create_actiontable(number_of_terminals, number_of_states : Int; action_data : Array_2[Int]) :
Array_2[Action] is
   (for row in [0 to (number_of_states-1)]
   cross column in [1 to (number_of_terminals*2) step 2] do
   entry =
      case action_data[row, column] of
         0 : Perror;
         1 : Shift(action_data[row, column + 1]);
         2 : Reduce(action_data[row, column + 1]);
         3 : Accept;
      otherwise:
         null:Action;   ** never occurs, but won't work without otherwise:
      esac;
   return all
      entry
   od).set_lower_1(0)

class Parser(lexer : Automaton)
{
   initial_state = (null:yytoken, 0)::null:Stack;

   number_of_states = 38;

   ** array of Ints representing each rules lhs nonterminal for the augmented grammar
   lhs_of_rule = [1..12 |
0,1,1,2,3,3,3,3,3,3,3,3];

   ** An Action, in the action table is represented with a pair
   ** e.g. (a, v), where 'a' represents the action and 'v' the integer
   ** value associated with the action, e.g. a state or a rule number.
   action_data =
[0..37, 1..22 |
2,2,0,0,0,0,0,0,1,1,0,0,1,2,0,0,0,0,1,3,0,0,
0,0,0,0,0,0,0,0,1,20,0,0,1,21,0,0,0,0,1,22,0,0,
0,0,0,0,0,0,0,0,1,1,0,0,1,2,0,0,0,0,1,3,0,0,
0,0,2,5,2,5,2,5,0,0,0,0,2,5,2,5,2,5,0,0,0,0,
3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,1,17,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,2,4,1,7,1,8,0,0,0,0,1,9,1,10,1,11,0,0,0,0,
0,0,0,0,0,0,0,0,1,1,0,0,1,2,0,0,0,0,1,3,0,0,
0,0,0,0,0,0,0,0,1,1,0,0,1,2,0,0,0,0,1,3,0,0,
0,0,0,0,0,0,0,0,1,1,0,0,1,2,0,0,0,0,1,3,0,0,
0,0,0,0,0,0,0,0,1,1,0,0,1,2,0,0,0,0,1,3,0,0,
0,0,0,0,0,0,0,0,1,1,0,0,1,2,0,0,0,0,1,3,0,0,
0,0,2,10,2,10,2,10,0,0,0,0,2,10,2,10,1,11,0,0,0,0,
0,0,2,9,2,9,2,9,0,0,0,0,2,9,2,9,1,11,0,0,0,0,
```

```
0,0,2,8,2,8,1,8,0,0,0,0,2,8,1,10,1,11,0,0,0,0,
0,0,2,7,2,7,2,7,0,0,0,0,2,7,2,7,1,11,0,0,0,0,
0,0,2,6,2,6,1,8,0,0,0,0,2,6,1,10,1,11,0,0,0,0,
2,2,0,0,0,0,0,0,1,1,0,0,1,2,0,0,0,0,1,3,0,0,
2,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,2,12,2,12,2,12,0,0,0,0,2,12,2,12,2,12,0,0,0,0,
0,0,0,0,0,0,0,0,1,20,0,0,1,21,0,0,0,0,1,22,0,0,
0,0,0,0,0,0,0,0,1,20,0,0,1,21,0,0,0,0,1,22,0,0,
0,0,0,0,2,5,2,5,0,0,2,5,2,5,2,5,2,5,0,0,0,0,
0,0,0,0,1,24,1,25,0,0,1,26,1,27,1,28,1,29,0,0,0,0,
0,0,0,0,0,0,0,0,1,20,0,0,1,21,0,0,0,0,1,22,0,0,
0,0,0,0,0,0,0,0,1,20,0,0,1,21,0,0,0,0,1,22,0,0,
0,0,2,11,2,11,2,11,0,0,0,0,2,11,2,11,2,11,0,0,0,0,
0,0,0,0,0,0,0,0,1,20,0,0,1,21,0,0,0,0,1,22,0,0,
0,0,0,0,0,0,0,0,1,20,0,0,1,21,0,0,0,0,1,22,0,0,
0,0,0,0,0,0,0,0,1,20,0,0,1,21,0,0,0,0,1,22,0,0,
0,0,0,0,2,10,2,10,0,0,2,10,2,10,2,10,1,29,0,0,0,0,
0,0,0,0,2,9,2,9,0,0,2,9,2,9,2,9,1,29,0,0,0,0,
0,0,0,0,2,8,1,25,0,0,2,8,2,8,1,28,1,29,0,0,0,0,
0,0,0,0,2,7,2,7,0,0,2,7,2,7,2,7,1,29,0,0,0,0,
0,0,0,0,2,6,1,25,0,0,2,6,2,6,1,28,1,29,0,0,0,0,
0,0,0,0,2,12,2,12,0,0,2,12,2,12,2,12,2,12,0,0,0,0,
0,0,0,0,1,24,1,25,0,0,1,37,1,27,1,28,1,29,0,0,0,0,
0,0,0,0,2,11,2,11,0,0,2,11,2,11,2,11,2,11,0,0,0,0];

   action_table = create_actiontable(11, number_of_states, action_data);

   goto_table =
[0..37, 1..3 |
4,5,6,
0,0,23,
0,0,19,
0,0,0,
0,0,0,
0,0,0,
0,0,0,
0,0,16,
0,0,15,
0,0,14,
0,0,13,
0,0,12,
0,0,0,
0,0,0,
0,0,0,
0,0,0,
0,0,0,
18,5,6,
0,0,0,
0,0,0,
0,0,36,
0,0,35,
0,0,0,
0,0,0,
0,0,34,
0,0,33,
0,0,0,
0,0,32,
0,0,31,
0,0,30,
0,0,0,
```

```
0,0,0,
0,0,0,
0,0,0,
0,0,0,
0,0,0,
0,0,0,
0,0,0];

** class header user code.


}

   yyparse : Bool is
      initially
         stack = initial_state;
         a = self.next_token;
         exit = false;
      while not exit do
         (tok, s) = stack.head;
         (new exit, new stack, new a) =
         typecase action = action_table[s, self.terminal_number(a)] of
            Shift  :  (false, (a, action.i)::stack, self.next_token);
            Reduce :  {
                         rule_number = action.j;
                         nt_type = lhs_of_rule[rule_number];
                         ** execute the user's action.
                         ** and perform the reduction by popping entries from the stack
                         ** Finally generate synthesised attribute.
                         (lhs, new_stack) = self.execute_production_rule(stack, rule_number);

                         (tok_top, s_top) = new_stack.head;

                         return
                            (false, (lhs, goto_table[s_top, nt_type])::new_stack, a)
                      };
            Accept :  (true, stack, a);
            Perror  : self.parse_error;
         esac;

         return
            exit
      od

   next_token : yytoken is
      lexer.yylex

   ** currently only reports failure.
   ** Error recovery not yet implemented.
   parse_error : (Bool, Stack, yytoken) is
   {
      os!stderr("Parse failed. - Error recovery not yet implemented.\n");
      return
         (false, null:Stack, null:yytoken)
   }

   ** convert terminal symbols returned by yylex into values suitable for
   ** indexing action tables with.
   terminal_number (symb : yytoken) : Int is
      typecase symb of
```

```
         EOF: 1;
         Newline: 2;
         Plus: 3;
         Mult: 4;
         Leftp: 5;
         Rightp: 6;
         Minus: 7;
         Divide: 8;
         Carot: 9;
         Number: 10;
         UMINUS: 11;
      esac

   pop_stack(stack : Stack) : (yytoken, Stack) is
   {
      (tok, s) = stack.head;
      return
         (tok, stack.tail)
   }

   ** perform user action according to 'rule_number' and reduce stack
   ** organise synthesised attribute calculations
   execute_production_rule(s1 : Stack; rule_number : Int) : (yytoken, Stack) is
      case rule_number of
         1: (null:yytoken, s1);
         2: {
               user_result = {return true };
               return
                  (list(user_result), s1)
         };
         3: {
               (a1, s2) = self.pop_stack(s1); list1 = typecase a1 of list: a1.yylval; esac;
               (a2, s3) = self.pop_stack(s2); Newline1 = typecase a2 of Newline: a2.yylval; esac;
               (a3, s4) = self.pop_stack(s3); printline1 = typecase a3 of printline: a3.yylval;
esac;
               user_result = {return true };
               return
                  (list(user_result), s4)
         };
         4: {
               (a1, s2) = self.pop_stack(s1); expr1 = typecase a1 of expr: a1.yylval; esac;
               user_result = {os!message("= " ++ expr1.print ++ "\n"); return true};
               return
                  (printline(user_result), s2)
         };
         5: {
               (a1, s2) = self.pop_stack(s1); Number1 = typecase a1 of Number: a1.yylval; esac;
               user_result = {return Number1};
               return
                  (expr(user_result), s2)
         };
         6: {
               (a1, s2) = self.pop_stack(s1); expr2 = typecase a1 of expr: a1.yylval; esac;
               (a2, s3) = self.pop_stack(s2); Plus1 = typecase a2 of Plus: a2.yylval; esac;
               (a3, s4) = self.pop_stack(s3); expr1 = typecase a3 of expr: a3.yylval; esac;
               user_result = {return expr1 + expr2};
               return
                  (expr(user_result), s4)
         };
         7: {
```

```
            (a1, s2) = self.pop_stack(s1); expr2 = typecase a1 of expr: a1.yylval; esac;
            (a2, s3) = self.pop_stack(s2); Mult1 = typecase a2 of Mult: a2.yylval; esac;
            (a3, s4) = self.pop_stack(s3); expr1 = typecase a3 of expr: a3.yylval; esac;
            user_result = {return expr1 * expr2};
            return
               (expr(user_result), s4)
         };
         8: {
            (a1, s2) = self.pop_stack(s1); expr2 = typecase a1 of expr: a1.yylval; esac;
            (a2, s3) = self.pop_stack(s2); Minus1 = typecase a2 of Minus: a2.yylval; esac;
            (a3, s4) = self.pop_stack(s3); expr1 = typecase a3 of expr: a3.yylval; esac;
            user_result = {return expr1 - expr2};
            return
               (expr(user_result), s4)
         };
         9: {
            (a1, s2) = self.pop_stack(s1); expr2 = typecase a1 of expr: a1.yylval; esac;
            (a2, s3) = self.pop_stack(s2); Divide1 = typecase a2 of Divide: a2.yylval; esac;
            (a3, s4) = self.pop_stack(s3); expr1 = typecase a3 of expr: a3.yylval; esac;
            user_result = {return Int(expr1 / expr2)};
            return
               (expr(user_result), s4)
         };
         10: {
            (a1, s2) = self.pop_stack(s1); expr2 = typecase a1 of expr: a1.yylval; esac;
            (a2, s3) = self.pop_stack(s2); Carot1 = typecase a2 of Carot: a2.yylval; esac;
            (a3, s4) = self.pop_stack(s3); expr1 = typecase a3 of expr: a3.yylval; esac;
            user_result = {return expo(expr1, expr2)};
            return
               (expr(user_result), s4)
         };
         11: {
            (a1, s2) = self.pop_stack(s1); Rightp1 = typecase a1 of Rightp: a1.yylval; esac;
            (a2, s3) = self.pop_stack(s2); expr1 = typecase a2 of expr: a2.yylval; esac;
            (a3, s4) = self.pop_stack(s3); Leftp1 = typecase a3 of Leftp: a3.yylval; esac;
            user_result = {return expr1};
            return
               (expr(user_result), s4)
         };
         12: {
            (a1, s2) = self.pop_stack(s1); expr1 = typecase a1 of expr: a1.yylval; esac;
            (a2, s3) = self.pop_stack(s2); Minus1 = typecase a2 of Minus: a2.yylval; esac;
            user_result = {return -expr1};
            return
               (expr(user_result), s3)
         };
      otherwise:
         (null:yytoken, null:Stack);   ** because case wants it
      esac

end
```

Appendix B

B.1 Regular expressions

Regular expressions are composed of ASCII characters, which represent themselves and the metacharacters shown in fig. B.1.1, which have special meaning in a pattern.

```
? * + | ( ) ^ $ / ; . = < > [ { " \
```

Figure B.1.1 Regular expression reserved metacharacters.

The syntax and semantics of regular expressions will be described in order of decreasing precedence (from the most tightly-binding operators to the most weakly-binding):

[ ] means match any character within the brackets. The hyphen "-" denotes a range of characters, e.g. [a-z] stands for any lower-case alphabetic character, and [0-9a-fA-F] stands for any hexadecimal digit. If the first character after the bracket is a circumflex "^", it changes the meaning to be that of the complement of the characters listed, e.g. [^abc] stands for any character except a, b, or c. Inside the brackets, only the symbols " \", "-", "^" and "]" are treated as metacharacters.
To include "-", put it first or last or after the initial complement command "^".
To include "^", put it anywhere but first.
To include "]", put it first or after a complement command "^". e.g. []] or [^]]
Alternatively, they can be specified literally by preceding them with a "\".

"." A dot matches any single character except the newline character "\n".

"\" A backslash followed by one of the reserved characters specifies that character literally. Special escape sequences are also available, inside or outside of square-brackets:

\ddd	- where ddd is a 3 digit decimal escape (0-255).
\b	- backspace
\n	- newline
\t	- tab

"..." Matches everything within the quotation marks literally. All metacharacters except "\" loose their meaning.

{name} Refers to a substitution of a regular definition 'name' (defined in the UFOLex definitions section).

() Any regular expression may be enclosed in parentheses for syntactic effect.

r* A postfix operator which matches zero or more repetitions of r.

r+ A postfix operator which matches one or more repetitions of r.

r? A postfix operator which matches zero or one occurrence of r.

r{n1,n2} A postfix repetitious specification which matches between n1 and n2 repetitions of the preceding expression r.

r{n1} A postfix repetitious specification which matches exactly n1 repetitions of the preceding expression r.

r{n1,} A postfix repetitious specification which matches at least n1 repetitions of the preceding expression r.

r1r2 Concatenation of expressions denotes concatenation of strings. This matches any string that results from the concatenation of one string that matches r1 with another string that matches r2.

r1 | r2 An infix operator which matches either r1 or r2.

r1 / r2 An infix operator which denotes lookahead by matching r1 only if it is followed by r2. Only one "/" is permitted per pattern and hence a pattern can not contain both a "/" and "$".

^r When the circumflex "^" occurs at the beginning of an expression r, that expression will only match strings that occur at the beginning of a line (right after

a newline character) or at the very beginning of the input.

r$ When the dollar sign $ occurs at the end of an expression r, that expression will only match strings that occur at the end of a line (right before a newline character).

<<EOF>> A special pattern which matches the end of a file.

B.2 A collected syntax of the UFOLex input program

To help clarify the exact syntactic structure of the UFOLex input program, below is an EBNF grammar description. 'CODE' is used as a shorthand to describe sections of UFO code. IDENTIFIER is defined as [A-Za-z][A-Za-z0-9'-]* using a regular expression syntax. The EBNF grammar description for regular expressions is rather complicated and hence a written description is also given.

```
goal := 'CODE' '%%' ufolex_definitions* '%%' rules+ 'EOF'

ufolex_definitions := IDENTIFIER '=' regular_expression ';'
                    | '%s' 'IDENTIFIER'+ ';'
                    | '%S' 'IDENTIFIER'+ ';'

rules := lexrule
       | state_cond lexrule
       | state_cond '(' lexrule+ ')'

lexrule := regular_expression '=>' 'CODE' ';'

state_cond : '<' IDENTIFIER [',' IDENTIFIER]+ '>'
```

The UFOLex regular expression syntax below, is complicated because it incorporates precedence of the operators. To begin with, we deal with left and right context sensitivity:

```
regular_expression := regexp
                    |  '^' regexp

regexp := exp
        |  exp '$'
        |  exp '/' exp
```

Next we deal with alternation and concatenation of patterns. An expression can consist of a number of alternative patterns and each pattern can consist of several singleton expressions concatenated together.

```
exp := pattern
    |  exp '|' pattern

pattern := singleton+
```

Each singleton expression can be composed of nested expressions or character classes, specified using a number of methods. Various 'repetition' operators are provided to assign meaning to the singleton expressions preceding them.

```
singleton := CHAR
          |  '(' exp ')'
          |  '"' CHAR+ '"'
          |  '.'
          |  '[' char_class+ ']'
          |  '[' '^' char_class+ ']'
          |  singleton '*'
          |  singleton '+'
          |  singleton '?'
          |  singleton '{' NUMBER ',' NUMBER '}'
          |  singleton '{' NUMBER ',' '}'
          |  singleton '{' NUMBER '}'

char_class := CHAR
           |  CHAR '-' CHAR
```

where CHAR is some ASCII character.

B.3 Translation of notational shorthands in regular expressions

1. r+ is defined as one or more occurrences of 'r' i.e. r+ = rr*. It has the same associativity and precedence as '*'.
2. r? is defined as zero or one occurrence of 'r' i.e. (r)? denotes the language L(r) ∪{ε}. In other words r? = r | ε.

3. [abc] denotes a new character class. It is equivalent to a | b | c.
4. [0-9] denotes an abbreviated character class. It is equivalent to 0 | 1 | 2 | 3 | . . . | 9.
5. [^abc] denotes an excluded character class. Given that {a,b,c}∈V, then [^abc] is defined as V - {a, b, c}
6. r{2, 5} denotes repetitious occurrences of r i.e. between two and five repetitions which is equivalent to rr(r | ε)(r | ε)(r | ε)
7. r{3,} denotes repetitious occurrences of r i.e. at least three repetitions which is equivalent to rrrr*
8. r{3} denotes repetitious occurrences of r i.e. exactly three occurrences which is equivalent to rrr
9. '.' denotes a character class of [^\n]. Given that {\n}∈V, then [^\n] is defined as V - {\n}
10. "abc" denotes the concatenation of a,b and c, where a,b and c are treated literally.
11. \x denotes the character x, where x is treated literally.
12. \ddd denotes the ASCII character with a decimal code of ddd, in the range 0 to 255.

B.4 BNF grammar for UFOLex programs

BNF grammar for parsing a lex program:

```
achieve_goal : USERCODE definitions_section rules_section
             ;
```

BNF grammar for parsing the definitions section:

```
definitions_section : LEXMARK
                    | definition LEXMARK
                    | definition definitions_section
                    ;

definition : LEXSTATES identifier_list SEMI
           | IDENTIFIER ASSIGN regular_expression SEMI
           ;

identifier_list : IDENTIFIER
                | IDENTIFIER identifier_list
                ;
```

BNF grammar description for regular expression syntax:

```
regular_expression : regexp
                   | CARAT regexp
                   | EOF_OP
                   ;

regexp : exp
       | exp DOLLAR
       | exp SLASH exp
       | exp SLASH exp DOLLAR
       ;

exp : pattern
    | exp BAR pattern
    ;

pattern : singleton
        | pattern singleton
        ;

singleton : CHARCLASS
          | LEFTP exp RIGHTP
          | INVCOMMA quoted_string INVCOMMA
          | LBRACE IDENTIFIER RBRACE
          | singleton STAR
          | singleton PLUS
          | singleton QMARK
          | singleton LBRACE NUMBER COMMA NUMBER RBRACE
          | singleton LBRACE NUMBER RBRACE
          | singleton LBRACE NUMBER COMMA RBRACE
          ;

quoted_string : CHARCLASS
              | quoted_string CHARCLASS
              ;
```

BNF grammar description for parsing the rules section:

```
rules_section : rules EOF
              | rules rules_section
              ;
```

```
rules : rule
      | STATE rule
      | STATE LEFTP rule_list RIGHTP
      ;

rule : regular_expression ARROW ACTION
     ;

rule_list : rule
          | rule_list rule
          ;
```

Each token in the grammar above, is written using capital letters. These are the sorts of tokens that you would expect a scanner to process. There are two exceptions to this - STATE and CHARCLASS.

In ML_Lex, CHARCLASS represents any character set e.g. [A-Za-z0-9]. It has the advantage that it reduces the size of the syntax tree representing a regular expression with several alternation operators. This has obvious advantages such as a smaller finite automaton and an algorithm which will execute slightly faster. UFOLex also adopts this idea. Of course, a character set could no longer be recognised using a parser because of the relationship between the parse tree and the syntax tree, so we can let the scanner deal with this.

STATE represents a list of comma separated start states. The scanner deals with this because it simplifies the parser, which is constructed from the grammar. The parser can return information in a much less verbose manner. Below is the complete list of tokens that should be handled by the UFOLex program scanner:

1. USERCODE
2. LEXSTATES
3. IDENTIFIER
4. CARAT
5. DOLLAR
6. BAR
7. RIGHTP
8. INVCOMMA
14. LEXMARK
15. SEMI
16. ASSIGN
17. EOF_OP
18. SLASH
19. CHARCLASS
20. LEFTP
21. LBRACE

9. RBRACE
10. PLUS
11. NUMBER
12. EOF
13. ARROW
22. STAR
23. QMARK
24. COMMA
25. STATE
26. ACTION

B.5 UFOLex's BNF grammar for recursive decent parsing

Below is a version of the grammar given in B.4 having been left factored and had left recursion eliminated from it. This grammar is suitable for implementation of a recursive decent parser.

BNF grammar for parsing a lex program:

```
achieve_goal : USERCODE definitions_section rules_section
             ;
```

BNF grammar for parsing the definitions section:

```
definitions_section : definition definitions_section2
                    | LEXMARK
                    ;

definitions_section2 : LEXMARK
                     | definitions_section
                     ;

definition : LEXSTATES identifier_list SEMI
           | IDENTIFIER ASSIGN regular_expression SEMI
           ;

identifier_list : IDENTIFIER identifier_list2
                ;

identifier_list2 : identifier_list2
                 |
                 ;
```

BNF grammar description for regular expression syntax:

```
regular_expression : regexp
                   | CARAT regexp
                   | EOF_OP
                   ;

regexp : exp regexp2
       ;

regexp2 : SLASH regexp3
        | DOLLAR
        |
        ;

regexp3 : exp regexp4
        ;

regexp4 : DOLLAR
        |
        ;

exp : pattern exp2
    ;

exp2 : BAR pattern exp2
     |
     ;

pattern : singleton pattern2
        ;

pattern2 : singleton pattern2
         |
         ;

singleton : CHARCLASS singleton2
          | LEFTP exp RIGHTP singleton2
          | INVCOMMA quoted_string INVCOMMA singleton2
          | LBRACE IDENTIFIER RBRACE singleton2
          ;

singleton2 : LBRACE NUMBER singleton3
           | STAR singleton2
```

```
            | PLUS singleton2
            | QMARK singleton2
            |
            ;

singleton3 : COMMA singleton4
            | RBRACE singleton2
            ;

singleton4 : NUMBER RBRACE singleton2
            | RBRACE singleton2
            ;

quoted_string : CHARCLASS quoted_string2
              ;

quoted_string2 : CHARCLASS quoted_string2
               |
               ;
```

BNF grammar description for parsing the rules section

```
rules_section : rules rules_section2
              ;

rules_section2 : EOF
               | rules_section
               ;

rules : STATE rules2
      | rule
      ;

rules2 : rule
       | LEFTP rule_list RIGHTP
       ;

rule : regular_expression ARROW ACTION
     ;

rule_list : rule rule_list2
          ;
```

```
rule_list2 : rule rule_list2
           |
           ;
```

Appendix C

C.1 A collected syntax of the UFOYacc input program

This section gives a collected syntax description of the UFOYacc input program in a BNF form. The general UFOYacc input program grammar is of the following form:

```
Supporting UFO Code
%%
Declarations
%%
Translation Rules
```

Below is a list of the tokens used by the subsequent BNF grammar for describing each section of the input program.

```
USERCODE:          "{........}"
TYPE:              "<...>"
LEXMARK:           "%%"
TERMINALS:         "%term"
NONTERMINALS:      "%nonterm"
LEFTASSOC:         "%left"
RIGHTASSOC:        "%right"
NONASSOC:          "%nonassoc"
START:             "%start"
PRECEDENCE:        "%prec"
COLON:             ":"
SEMI:              ";"
BAR:               "|"
SYMBOL:            [A-Za-z0-9_']
EOF:               <<EOF>>
```

Now follows the concrete syntax of the input program in a BNF form,

```
goal : USERCODE declarations translation_rules
     ;

declarations : LEXMARK
             | decl declarations
             ;
```

```
decl : TERMINALS TYPE symbol_list
     | NONTERMINALS TYPE symbol_list
     | LEFTASSOC symbol_list
     | RIGHTASSOC symbol_list
     | NONASSOC symbol_list
     | START SYMBOL
     ;

symbol_list : SYMBOL
            | symbol_list SYMBOL
            ;

nonterminal : SYMBOL
            ;

terminal : SYMBOL
         ;

translation_rules : EOF
                  | rule
                  | translation_rules rule
                  ;

rule : nonterminal COLON alternatives
     ;

alternatives : right_hand_side SEMI
             | right_hand_side BAR alternatives
             ;

right_hand_side : member choose_prec USERCODE
                | member right_hand_side
                ;

choose_prec : PRECEDENCE terminal
            |
            ;

member : terminal
       | nonterminal
       | ;
```

C.2 UFOYacc's BNF grammar for recursive decent parsing

```
goal : USERCODE declarations translation_rules
     ;

declarations : LEXMARK
             | decl declarations
             ;

decl : TERMINALS TYPE symbol_list
     | NONTERMINALS TYPE symbol_list
     | LEFTASSOC symbol_list
     | RIGHTASSOC symbol_list
     | NONASSOC symbol_list
     | START SYMBOL
     ;

symbol_list : SYMBOL symbol_list2
            ;

symbol_list2 : SYMBOL symbol_list2
             |
             ;

nonterminal : SYMBOL
            ;

terminal : SYMBOL
         ;

translation_rules : rule translation_rules2
                  ;

transition_rules2 : translation_rules
                  | EOF
                  ;

rule : nonterminal COLON alternatives
     ;

alternatives : right_hand_side alternatives2
             ;
```

```
alternatives2 : SEMI
              | BAR alternatives
              ;

right_hand_side : member right_hand_side2
                ;

right_hand_side2 : choose_prec USERCODE
                 | right_hand_side
                 ;

choose_prec : PRECEDENCE terminal
            |
            ;

member : terminal
       | nonterminal
       |
       ;
```

Appendix D

D.1 Lex and Yacc - further motivations.

Hand-written programs to perform lexical analysis and syntactic analysis tend to have several draw backs:

1. Hard to construct for complex languages
2. Difficult to maintain/update
3. Difficult to verify that the program recognises the correct grammar/tokens
4. Time consuming to construct

Lex and Yacc successfully solve the general problem of lexical and syntactic analysis, respectively. They:

1. Save the programmer time due to simplification of the task and increased maintainability.
2. Are good for people who are not experts in pattern matching techniques.
3. Guarantee that the language recognised by the compiler is identical to that defined by the grammar.
4. Provide a rigid separation of lexing and parsing leading to a simpler design of one or other of these phases, a cleaner overall language design, enhanced compiler compatibility - Input alphabet and device specific peculiarities are restricted to the lexical analyser and finally improved performance due to the utilization of efficient algorithms for the specific task.

Appendix E

E.1 Regular grammar property

The languages recognised by finite automata, are said to conform to regular grammars. A grammar is said to be regular if it consists entirely of productions of the form:

A -> e
A -> a
A -> aB

By convention, capital letters in definitions represent non-terminals and lowercase letters represent terminals. A regular grammar has a maximum of one non-terminal on the right hand side of a production. Each derivation of a string in the language described by the regular grammar, is entirely determined by the derived sequence itself i.e. more than one derivation for the same derived string is impossible, unlike other context-free grammars. Hence a regular expression can never specify an ambiguous grammar.

Regular expressions are a notation used to describe regular languages of this exact type.

E.2 Left context

There are three ways that left context sensitivity can be exploited in UFOLex:

1. '^' character. This matches a pattern only at the beginning of a line e.g. after '\n'. '^' doesn't match characters, it just specifies context.
2. Start States can be used to require that one matched string precedes another.
3. Explicitly in action code. By setting flags to pass context information from one token's action to that of another.

E.3 Right context

There are two ways that right context sensitivity can be exploited in UFOLex:

1. '$' character. This matches a pattern only at the end of a line, e.g. before '\n'. '$' doesn't match characters, it just specifies context. It is equivalent to '/\n' and thus

cannot be used in conjunction with the trailing context operator '/'.

2. '/' the explicit trailing context operator. '/' doesn't match any characters. UFOLex only matches the pattern with the longest trailing context.

Appendix F

F.1 UFO1.0/S compiler problems

Parser Failed!

- "Classes with more than 20 formal parameters/instance values not supported by current implementation"

- Parser gives up when parsing large array initialisations due to heap overflows.

- Very slow parsing time when compiling large array initialisations and generates C code which is also very slow to compile.

- Parser gives up when parsing large strings due to stack overflows.

Passing arrays

- Multiple updates of an array, especially when passed as a parameter to a recursive function, results in large garbage collection overheads giving rise to heap overflows and a significant decrease in performance.

Executable Crashes - controlled

- It is necessary to re-write algorithms to avoid spurious deadlocks so strictly speaking, parallelism isn't entirely "*exploited without the programmer having to expend any effort*". e.g updating stateful symbol tables during recursive decent parsing.

Executable Crashes - uncontrolled (coredump)

- Non-pointer types in Generics fail leading to either uncontrolled crashes or confusing program behaviour. For example a generic Set class works with sets of strings but fails with ints, floats, bools, chars - without warning.

- Type system sometimes becomes confused/unstable as a result of underlying generic implementation problems.

Compiler Crashes (coredump)

- If a class attribute's name is accidentally redefined within the same class, in the form of a parameter to a method, then the UFO1.0/S compiler crashes. This is hard to diagnose.

Compiling Process

- Slow to compile the generated C code.

- Make system dependencies are incomplete and often the only remedy is to force a total re-compilation.

- Certain error reports don't give sufficient information to track down errors, especially when generics are involved.

- Variables called 'tmp' in UFO cause type case errors when UFO compiler begins to compile the C.

- '.ufo' module names cannot include dashes, e.g. "in-post.ufo" because otherwise when the C compilation takes place, errors relating to bad #includes are formed.